Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 55 (2005) 117–159
Science of Computer Programming
www.elsevier.com/locate/scico

Live and let die: LSC based verification of UML models

Werner Damm*, Bernd Westphal

Department für Informatik, Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, Germany

Received 31 August 2003; received in revised form 15 April 2004; accepted 30 May 2004
Available online 11 November 2004

Abstract

This paper addresses the problem of formal verification of UML models in the semantics of Damm and Josko et al. (Science of Computer Programming, this issue). The problem is twofold in that it requires on the one hand a specification language which is rich enough to express properties about entities that are only created during a run of the system and on the other hand a means to abstract the a priori unbounded state space to a finite one which lends itself to treatment by approved finite state methods.

As the specification language, the paper proposes to extend Live Sequence Charts as presented by W. Damm and D. Harel [LSCs: breathing life into message sequence charts, Formal Methods in System Design 19 (1) (2001) 121–141] and J. Klose [Live sequence charts: A graphical formalism for the specification of communication behavior, Ph.D. Thesis, Carl von Ossietzky Universität Oldenburg, 2003] by means of dynamically bound instance lines and equips it with a formal semantics w.r.t. the UML domain.

For verification, the paper proposes to transfer to the UML domain the methodology of K.L. McMillan [A methodology for hardware verification using compositional model checking, Science of Computer Programming 37 (2000) 279–309], comprising a first step which is based on results of C.N. Ip and D.L. Dill [Better verification through symmetry, Formal Methods in System Design 9 (1–2) (1996) 41–75] about symmetric data-types and for which F. Xie and J.C. Browne [Integrated state space reduction for model checking executable object-oriented software system designs, in: R.-D. Kutsche, H. Weber (Eds.), FASE, Lecture Notes in Computer Science, vol. 2306,

* Corresponding address: OFFIS, Oldenburg, Germany.
E-mail addresses: damm@informatik.uni-oldenburg.de (W. Damm), westphal@informatik.uni-oldenburg.de (B. Westphal).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2004.05.013

Springer, 2002] coined the term "Query Reduction" and, as second step, an abstract interpretation called "data-type reduction" to construct a finite state over-approximation of the original model for each query. The paper also briefly discusses counter-measures against false-negatives occurring in the over-approximation.

1. Introduction and related work

The increasing use of UML or specialised sublanguages thereof in the domain of safety-critical systems design raises a need for rigorous techniques for formally verifying UML models against requirements specifications.

The pre-requisites necessary to provide such techniques based on approved finite state methods are threefold: first, we need a formal semantics of UML (or an adequate sublanguage); second, we need a specification language on the level of the UML model which is able to express properties about objects that are only created during a run of the system and which provides a means to explicitly specify object creation and destruction; and third, we need a methodology to reduce the in general infinite state space of UML models due to a priori non-existent upper bounds on the numbers of objects created during a run of the system.

We satisfy the first pre-requisite by using the UML sublanguage and semantics of [7]. They basically consider (possibly active) classes whose behaviour is given by flat, i.e. non-nested, statecharts. The semantics of a UML model in the definition of [7] is a Symbolic Transition System (STS). Each state of the transition system is a finite number of unbounded arrays of records, one for each class.

As a specification language for inter-object communication we propose an extension of the Live Sequence Charts [4] language (LSC) where we interpret instance lines as free logical variables which are meant to be bound to entities of the system in each state of a run. We equip them with a UML specific mapping, i.e. explain in terms of the STS defined by [7], what it means to, for example, send or receive an event as shown in the LSC.

In order to be able to employ finite state verification methods, we propose a two-step approach exploiting symmetries in the state space of the transition system representing the formal semantics of a UML model. In the first step, we observe that the infinite number of bindings of objects to instance lines is implied by a finite, representative set of bindings. Each binding from this set can be verified separately for the as yet unbounded state space. In the second step, we apply, depending on the particular binding, a data-type reduction, an abstraction which yields a finite over-approximation of the original system.

1.1. Specification languages and formal verification for UML

The problem of an *adequate specification language* for systems with dynamic creation and destruction of entities has been addressed in [28,14,2,41,9]. The authors of [28,14] present an alternative approach to explain binding of instances to instance lines of an LSC with the same underlying intuition, but in addition to allow one to quantify single instance lines. The description is tailored for application in the play-in/play-out tool [14], that is,

for observing or "playing-out" a complete system. In contrast, our approach is chosen s.t. the theories of symmetry reduction and data-type reduction apply for yielding finite state verification tasks.

From the field of Java verification, the approach of temporal logic patterns in the textual Bandera Specification Language (BSL) [2] is closely related. It lacks expressiveness compared to LSCs since it is restricted to a rather small set of patterns. A BSL pattern is basically a fixed temporal logic formula over arbitrary predicates and universally quantified variables of class type that are bound at runtime.

The Evolution Temporal Logic (ETL) [41] is introduced to express properties of systems which make use of a general heap structure, i.e. a means to dynamically allocate memory without static names for the memory locations. The logic provides a quantification over all instances of a type, operators to denote the point in time of creation and destruction, and a transitive-closure operator in order to express reachability between objects by navigation expressions. For ETL verification, [41] proposes the static analysis technique of shape analysis which applies the abstract interpretation of a given program to an (always finite) shape graph representing the possible runs of the program until a fixed point is reached.

The Allocational Temporal Logic (ATL) [9] can be seen as a predecessor of ETL that abstracts from everything but allocation operations and identities of instances, and thus in particular abstracts from the data part of a system. It is intended to express properties over Allocational Büchi Automata as introduced in [9] together with a model-checking algorithm.

The problem of *automatic formal verification of a significant sublanguage of UML* by standard finite state methods has been addressed by various authors at different levels of sophistication [8,25,3,40,36,38,12,35,26]. The works of [8,25,3,38,12,35,26] concentrate on different subsets of UML's state-machine language and consider only a single object or an explicitly given finite set of objects, i.e. do not address dynamic creation or destruction of objects. The specification languages used in these approaches range from temporal logic expressions over variable names on the level of the underlying model-checker's input language [38,35,26] to temporal logic (or patterns) on the UML model level [40,12].

The tools presented in [25,36] employ, besides a standard temporal logic, UML Collaboration Diagrams to specify a desired scenario and provide an automatic search for a run which exhibits that the given scenario can be observed (this corresponds to existential verification of LSCs), but they require a static relation between the objects in the Collaboration Diagram and those in the system. UML Collaboration Diagrams are closely related to the UML Sequence Diagram language which is covered by LSCs; thus they are as regards expressiveness a proper sublanguage of LSCs.

Recent achievements of [40] implement object creation and destruction explicitly in the input language of the formal verification tool employed based on "switching on and off" objects like in [7] or, as in [33], translate the UML model into an intermediate language which provides constructs for this kind of dynamics s.t. the problem of choosing a finite representation is shifted to the translation step from the intermediate language to the formal verification tool employed. Both approaches presuppose a finite bound on the number of objects alive in each snapshot of a run as long as the target is a finite state formal verification tool.

1.2. Structure

The remainder of the paper is organised as follows. In Section 2 we introduce signatures, expressions, and interpretations in order to be able to define symbolic transition systems [27], our computational model, and linear temporal logic (LTL) over expressions. Section 3 recalls the main motivations for the introduction of LSCs and provides the general semantics of LSCs without timers as quantified LTL formulae. The LSC language is then specialised for the context of UML in terms of [7]. In Section 4 we give a brief survey of the original literature on exploiting system symmetries for formal verification, provide the theory of query reduction, contribute the as yet neglected proofs, and show how it applies to LSCs in the context of UML. Section 4.3 presents the theory of data-type reduction together with as yet missing proofs and discusses the common class of "interference" false-negatives caused by the data-type reduction abstraction and how, generalising the methodology of [32], interference could be avoided by separately proving and then assuming non-interference lemmata derived from information in the UML model. Section 5 concludes.

2. Preliminaries

As our computational model, we take symbolic transition systems (STS) [27], which allow a purely syntactical description of a transition system by first-order-logic expressions over a signature. Section 2.2 defines an STS as two first-order-logic predicates over a signature, so we first introduce signatures, predicate- and first-order-logic expressions and their interpretation in Section 2.1. Section 2.3 defines linear temporal logic and the satisfaction of LTL formulae by a symbolic transition system in order to make it possible to explain the semantics of Live Sequence Charts in the following section and to provide the formal foundation of the proofs in Section 4.

All definitions are standard; hence the reader may safely skip this section on the first reading.

2.1. Predicate- and first-order-logic expressions

Definition 1 (*Predicate-logic Expressions*). Let V be a set of typed variables and Ω a set of typed constants. The pair $B = (V, \Omega)$ is called the *signature*. The set $Expr(B)$ of typed *expressions over* B is defined inductively as follows:

(i) Let $v \in V$ be a variable of unstructured type τ, record type $\tau_1 \times \cdots \times \tau_n$, $n \in \mathbb{N} = \{1, 2, \ldots\}$, or array type $\tau_1 \to \tau_2$.
Then v is an expression over B of type τ, $type(v) =_{df} \tau$, of type $\tau_1 \times \cdots \times \tau_n$, $type(v) =_{df} \tau_1 \times \cdots \times \tau_n$, or of type $\tau_1 \to \tau_2$, $type(v) =_{df} \tau_1 \to \tau_2$.

(ii) Let $f \in \Omega$ be a constant of type $\tau_1 \times \cdots \times \tau_n \to \tau$, $n \in \mathbb{N}_0 = \mathbb{N} \cup \{0\}$. Let $expr_i \in Expr(B)$, $type(expr_i) = \tau_i$, for $0 < i \leq n$. Then $expr = f(expr_1, \ldots, expr_n)$ is an expression over B of *arity* n, $type(expr) =_{df} \tau$.
A constant f of type $\tau_1 \times \cdots \times \tau_n \to \mathbb{B} = \{true, false\}$ is called a *predicate*.[1]

[1] The symbol $\mathbb{B}$ will be used to denote both the type and its domain $\mathcal{D}_{\mathbb{B}}$.

We use T_B to denote the set of all types of variables and constants in B. The elements of the set $Expr_{PL}(B) =_{df} \{expr \in Expr(B) \mid type(expr) = \mathbb{B}\}$ are called *predicate-logic expressions over B*. □

In the following, we assume $\Omega \supseteq \{true, \vee, \neg, false, \wedge, \dot{\vee}, \Longrightarrow, \Longleftrightarrow\}$ for each signature, where the symbols '*false*', '$\wedge$', '$\dot{\vee}$' (exclusive or), '$\Longrightarrow$', and '$\Longleftrightarrow$' are used as abbreviations with the conventional definition for brevity.

Definition 2 (*First-order-logic Expressions*). Let $B = (V, \Omega)$ be a signature. The set $Expr_{FO}(B)$ of *first-order-logic (FOL) expressions over B* is defined inductively as follows:

(i) Let $expr \in Expr_{PL}(B)$ be a predicate-logic expression. Then '*expr*' is a first-order-logic expression.

(ii) Let $expr, expr_1, expr_2 \in Expr_{FO}(B)$ be first-order-logic expressions and $\tau \in T_B$ a type. Then '$\exists\ x \in \tau : expr$', '$\neg expr_1$', and '$expr_1 \vee expr_2$' are first-order-logic expressions of type $\mathbb{B}$.
Each occurrence of the variable $x \in V$ in '*expr*' is called *bound*.

Let $expr \in Expr_{FO}(B)$. A variable $x \in V$ occurring in '*expr*' is called *free in 'expr'* if not all occurrences are bound.

We call a first-order-logic expression over $B = (V \cup V', \Omega)$, that is, an expression referring to both unprimed and primed versions of the variables in V, a *first-order-logic transition predicate over B*. □

In the following, we use the conventional definition of the symbols '$\forall$', '$\wedge$', '$\dot{\vee}$', '$\Longrightarrow$', and '$\Longleftrightarrow$' in first-order-logic expressions for brevity.

Definition 3 (*Interpretation*). Let $B = (V, \Omega)$ be a signature, $\mathcal{D} \supseteq \bigcup_{\tau \in T_B} \mathcal{D}_\tau$ a *domain* for all types used in B, and $\mathcal{I} : \Omega \to \bigcup_{n \in \mathbb{N}_0} \{(\times_{i=0}^{n} \mathcal{D}_{\tau_i}) \to \mathcal{D}_\tau \mid \tau_1, \ldots, \tau_n, \tau \in T_B\}$ an *interpretation of the constants* which assigns to each constant $f \in \Omega$ of type $\tau_1 \times \cdots \times \tau_n \to \tau$ a value $\mathcal{I}(f) : \mathcal{D}_{\tau_1} \times \cdots \times \mathcal{D}_{\tau_n} \to \mathcal{D}_\tau$. The tuple $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ is called a *structure of B*.

A function $s : V \to \mathcal{D}$ is called a *(type-consistent) valuation* of V if it assigns to each variable $v \in V$ of type τ a value $s(v) \in \mathcal{D}_\tau$. The set of valuations is called Σ_B (or Σ if the signature is clearly determined by the context).

We use $\mathcal{M}[\![expr]\!](s)$ to denote the canonical *interpretation* of the first-order-logic expression '*expr*' in the valuation s. □

The interpretation $\mathcal{M}[\![expr]\!](s, s')$ of a transition predicate is defined analogously, letting s provide the interpretation of unprimed and s' the interpretation of primed variables in *expr*.

In the following, we consider only interpretations $\mathcal{M}$ which give the canonical interpretation to the constants '*true*', '$\vee$', and '$\neg$'.

2.2. Symbolic transition systems

Definition 4 (*STS*). A *symbolic transition system* (STS) $S = (B, \Theta, \rho)$ consists of $B = (V, \Omega)$, a signature with a finite set V of variables, $\Theta \in Expr_{FO}(B)$, and ρ, a FOL transition

predicate over B. The set of variables $v \in V$ which are free in Θ or ρ are called *system variables* of S. □

An STS *induces* a transition system on the set of valuations of its system variables as follows:

Definition 5 (*Runs of an STS*). Let $S = (B, \Theta, \rho)$ be an *STS* and $\mathcal{M}$ a structure of B.

(i) A valuation $s \in \Sigma_B$ of the system variables of S is called a *snapshot of* S.

(ii) A snapshot $s \in \Sigma_B$ of S is called *initial* iff $\mathcal{M}[\![\Theta]\!](s) = \mathit{true}$.

(iii) Let $s, s' \in \Sigma_B$ be snapshots of S.
Snapshot s' is called the *S-successor* of s iff $\mathcal{M}[\![\rho]\!](s, s') = \mathit{true}$.

(iv) A *computation* or *run* of S is an infinite sequence of snapshots,
$r = s_0\, s_1\, s_2 \ldots$, satisfying the following requirements:
- *Initiation:* s_0 is initial.
- *Consecution:* Snapshot s_{j+1} is an S-successor of s_j, for each $j \in \mathbb{N}_0$.

(v) The set of all computations of S is called *runs*(S).
For a run $r \in \mathit{runs}(S)$ we use r^i, $i \in \mathbb{N}_0$, to denote the i-th snapshot and

$$r/i =_{df} r^i\, r^{i+1}\, r^{i+2} \ldots$$

to denote the infinite sequence starting at r^i, $i \in \mathbb{N}_0$. □

2.3. Linear time logic

Definition 6 (*LTL*). Let B be a signature. An *LTL formula* over B is defined inductively as follows:

(i) $\mathit{expr} \in \mathit{Expr}_{PL}(B)$ is an LTL formula,

(ii) $\neg f$ and $f \vee g$ are LTL formulae if f and g are LTL formulae, and

(iii) $\mathbf{X}\, f$ ("next f"), $\mathbf{G}\, f$ ("globally f"), and $f\, \mathbf{U}\, g$ ("f until g") are LTL formulae if f and g are LTL formulae. □

In the following we define what it means for a run of an STS to satisfy an LTL formula and in addition introduce the orthogonal notions of existential versus universal and initial versus invariant satisfaction of an LTL formula. We will use this definition as the foundation of LSCs without timers in Section 3 instead of their Timed Büchi Automaton semantics [22] s.t. the theory of Section 4 applies directly.

Definition 7 (*Satisfaction of an LTL Formula*). Let $S = (B, \Theta, \rho)$ be an STS, ϕ an LTL formula over B, and $\mathcal{M}$ a structure of B. Let r be a (suffix of a) run of S.

We say r *satisfies* ϕ w.r.t. $\mathcal{M}$, denoted by $r \models_{\mathcal{M}} \phi$, iff:

(i) $\phi \equiv \mathit{expr}$ and $\mathcal{M}[\![\mathit{expr}]\!](r^0) = \mathit{true}$, or
(ii) $\phi \equiv f \vee g$ and $r \models_{\mathcal{M}} f$ or $r \models_{\mathcal{M}} g$, or
(iii) $\phi \equiv \neg f$ and $r \not\models_{\mathcal{M}} f$, or
(iv) $\phi \equiv \mathbf{X}\, f$ and $r/1 \models_{\mathcal{M}} f$, or

(v) $\phi \equiv \mathbf{G}\, f$ and $\forall\, i \in \mathbb{N}_0 : r/i \models_{\mathcal{M}} f$, or
(vi) $\phi \equiv f\, \mathbf{U}\, g$ and $\exists\, i \in \mathbb{N}_0 : r/i \models_{\mathcal{M}} g \wedge \forall\, 0 \leq j < i : r/j \models_{\mathcal{M}} f$.

We say S *existentially satisfies* ϕ *invariantly*, denoted by $S \models_{\mathcal{M},\exists} \phi$, iff

$$\exists\, r \in \mathit{runs}(S)\ \exists\, i \in \mathbb{N}_0 : r/i \models_{\mathcal{M}} \phi,$$

and *initially*, denoted by $S \models_{\mathcal{M},\exists,0} \phi$, iff $\exists\, r \in \mathit{runs}(S) : r/0 \models_{\mathcal{M}} \phi$.

We say S *universally satisfies* ϕ *invariantly*, denoted by $S \models_{\mathcal{M},\forall} \phi$, iff

$$\forall\, r \in \mathit{runs}(S)\ \forall\, i \in \mathbb{N}_0 : r/i \models_{\mathcal{M}} \phi,$$

and *initially*, denoted by $S \models_{\mathcal{M},\forall,0} \phi$, iff $\forall\, r \in \mathit{runs}(S) : r/0 \models_{\mathcal{M}} \phi$. □

3. Live sequence charts

Live Sequence Charts (LSC) are an extension of Message Sequence Charts (MSC), introduced to overcome serious deficiencies of the MSC language w.r.t. formal verification, so we begin with a short overview of the MSC language and the MSC dialect of UML Sequence Diagrams (SDs).

In Section 3.1, we recall the deficiencies of both formalisms; this is followed by a brief introduction of the subset of the LSC language which we consider as a specification language for formal verification of UML models for the scope of this paper. Section 3.2 provides a definition of general (domain-independent) LSCs that abstracts from syntactical aspects and from the elements which do not need a mapping, for example simultaneous regions that express *simultaneity* of elements. We do not elaborate on the temporal properties induced by the *relative position* or *partial ordering* of the elements of an LSC but take for granted that [22,24] provide us with a Timed Büchi Automaton that expresses just these *temporal* properties. Actually we do not consider this automaton but the equivalent LTL formula [37] that exists since we only consider *untimed* LSCs. In general, the formula refers to undefined constants ("placeholders") that are then given a domain-dependent interpretation that explains, for example, what it means to send or receive an asynchronous event.

In Section 3.3 we first define LSCs for UML models (in the sense of [7]) by giving constraints on the annotations of the LSC elements 'instance lines', 'event sending' and 'event reception', and 'conditions'. The satisfaction of an LSC by the UML model is then defined in terms of the UML model's so-called observer extension, binding objects to instance lines.

3.1. From message sequence charts and sequence diagrams to LSCs

The MSC language is a well-known visual formalism for describing behaviour of a system by visualising the *inter-entity* communication basically as arrows representing asynchronous messages between vertical instance lines representing entities within the system. Intuitively, the semantics of an MSC is a (partial) ordering in time of the observations of the messages occurring in it which is derived from the relative positions of the message arrows and their beginning or ending at instance lines.

The syntax and semantics of MSCs is standardised in different versions [19–21] that extend the core language by means to structure and compose MSCs in order to express loops and branches, by different annotations for timers and timing constraints, by means to explicitly state ordering information, and by different kinds of messages, e.g. synchronous messages which are for example in the UML domain interpreted as representing method calls and replies.

Although the MSC language was originally formalised in the telecommunication domain to match this domain's system specification language, it is not inherently bound to a particular domain, design language, or paradigm, but the kind of entities represented by an instance line can be chosen when giving semantics for a particular domain. Typical kinds of entities are processes in the context of process-oriented languages and objects in the object-oriented domain.

The Sequence Diagram language [34] of UML is an adoption of MSCs for UML where instance lines are in fact restricted to represent objects and where message types are provided to represent event based or method call communication.

The main deficiencies of MSCs and SDs w.r.t. their use in formal verification are that their usual interpretation is as a requirement for the system being able to perform the scenario shown in the MSC where one would like to also express that the system *always* behaves as depicted in the MSC, and that MSCs do not allow one to express *liveness* properties, i.e. to distinguish whether progress is enforced or not.

Furthermore, the MSC versions except for MSC-2000 do not allow one to specify an activation time; thus it is left open *when* a system has to show the behaviour described by the MSC in order to fulfil it. The intention of an MSC describing the error-handling behaviour, for example, is typically meant to be observed only after a particular error condition holds. No MSC version allows one to express this *activation* in terms of a sequence of messages, for example to express that error handling takes place after a sequence of a particular number of error events have been observed.

Other major drawbacks are related to conditions and simultaneity. Conditions annotated to locations on instance lines are merely comments up to MSC-2000 (and not even present in SDs) and simultaneity of items like messages and conditions cannot be expressed; hence it is not possible to e.g. specify that the system should be in a particular state when sending an event. Only MSC-2000 provides simultaneity, but restricted to pairs of a message and a timer. For a complete discussion of the sequence charts dialects and their shortcomings, the reader is referred to [22].

Note that, although LSCs also provide a more sophisticated semantical treatment of timers and time annotations in comparison to MSCs or SDs, we do not consider timers and time annotations at all in the following since the UML semantics of [7] which our presentation is based on is an untimed semantics.[2]

LSCs as introduced in [4] overcome the deficiencies of MSCs and SDs named above by employing the basic idea of distinguishing *mandatory* and *possible* behaviour per LSC element by introducing temperatures *hot* and *cold* for instance locations, messages,

[2] An extension of our approach to integer timers interpreted as counting steps of the underlying transition system is straightforward and used, e.g., in [22] for formal verification of synchronous Statemate designs.

and condition, and for the whole LSC by introducing the quantifications *existential* and *universal*. For an example see Fig. 2 which will be discussed in detail in Section 3.2.

Intuitively, an *existentially quantified* (possible) LSC is meant as a scenario, just like MSCs, i.e. it holds if there is a run of the system which complies to the LSC, while a *universally quantified* (mandatory) LSC requires that, whenever the LSC is activated by the activation condition and pre-chart, the system shows the behaviour depicted in the LSC body. Fig. 2 shows a universally quantified LSC as indicated by the solid frame around the LSC body; existential LSCs are drawn with a dashed frame.

The point in time when an LSC should be activated can be specified by giving a boolean *activation condition* and a (possibly empty) so-called *pre-chart* which is itself a restricted LSC. The LSC body is activated whenever the activation condition holds *and then* the behaviour depicted in its pre-chart is observed. Additionally, the activation of an LSC depends on the *activation mode* which can be one of 'initial', 'initial first', 'invariant', and 'iterative' [22]. In the following we only consider the activation modes which directly correspond to our Definition 7: 'initial', i.e. the LSC is activated at most once per run and only if its activation condition and pre-chart are observed from the initial step of a run on; and 'invariant', i.e. the LSC may be activated multiple times during a run, and there may even be overlapping activations. (Fig. 2 shows an LSC with a non-empty pre-chart, an activation condition denoted by AC and activation mode (AM) invariant.)

Within an LSC, each location, i.e. each place of an element on an instance line, e.g. a message start or end, is equipped with a temperature. A *hot* (mandatory) location enforces progress, that is, eventually the next location has to be reached. A *cold* (possible) location allows staying at the location forever, that is, the behaviour following a cold location need not be observed. Graphically, the temperature of a location ℓ is indicated by drawing the instance line segment between ℓ and its successor solid or dashed if ℓ is hot or cold respectively.

A *possible* or *cold* condition is a legal exit point of an LSC, i.e. if a run of the system adheres to the prefix of an LSC up to a cold condition and the condition does *not* hold, then the run is said to satisfy the LSC, since the LSC "exits" and is no longer activated. Reaching a location with a hot condition that does not hold is considered to be a violation of the specification. As an extension of conditions, LSCs also provide (possible or mandatory) local invariants, i.e. conditions which are not bound to a single location but to a start and end location, each inclusive or exclusive. For a complete presentation of the LSC features the reader is referred to [22].

In [4], the LSC language is introduced as a conservative extension of MSCs; thus LSCs are as domain, design language, and paradigm independent as MSCs. In particular, [22,24] give the formal semantics of LSCs independent from the *mapping*, and thus abstract from what "sending a message" actually means in a concrete system from a particular domain or what the "entities" bound to instance lines are. Only the ordering and temporal constraints expressed in the LSC are considered. We continue this effort by separating the syntactical annotation of an LSC from the final, domain specific semantics as presented in Fig. 1.

Thus for an application of the LSC language in the UML domain, firstly Definition 12 states that we call an LSC ranging over a UML model if its mapping adheres to certain well-formedness rules, e.g. the annotations of instance lines have to be of class type, the annotation of asynchronous messages have to be taken from the set of events in the model,

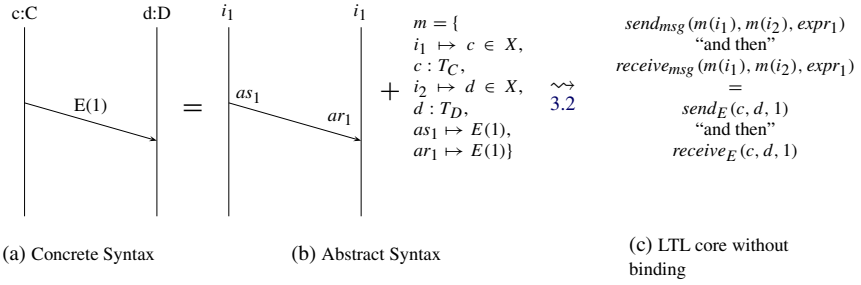


Fig. 1. **Domain specific mapping of an LSC:** The concrete syntax of an LSC (1(a)) is a representation for an unmapped LSC and a mapping function m (1(b)) that provides a syntactical annotation. Based on the mapping function we assume an LTL formula using constants $send_{msg}$ and $receive_{msg}$ that act as "placeholders" and get a domain specific interpretation (1(c)). For UML, for example, $send_{msg}$ is defined to refer to enqueueing of an event in Section 3.3. Furthermore, we have to define how c and d are bound. For UML, Section 3.3 chooses a quantification over all object identities. □

and the annotation of synchronous messages from the set of triggered operations. Secondly, definition Definition 14 provides an interpretation of the constants $send_{msg}$ and $receive_{msg}$ in terms of an observer extension of the UML model as defined in Definition 13. Which "entities" are to be bound to instance lines is already given by the *type* of the constants annotated to instance lines; in the case of UML it is a class type and thus we end up with a quantification over all objects.

The topic of binding of instance lines goes beyond the presentation of a specialisation of LSCs for the domain of Statemate designs as presented in [22] where the author requires an explicit static binding of instance lines to Statemate activities, which is possible since Statemate designs have a static structure, i.e. there is no dynamic creation or destruction of "system entities" as there is in the UML domain.

3.2. Live sequence charts

In general and independent from the design language domain, the intuition of an instance line within an LSC is the denotation of an entity of the system that the LSC is supposed to talk about. It depends on the domain what is considered an entity. If there are multiple instances of the same type of entity, then we want to take, for example, a universal LSC as an abbreviation for all possible *bindings* of concrete system entity instances to instance lines; thus instance lines can be seen as *free* or *logical* variables of the specification which are quantified over the entity's type.

In the following, we formalise this intuition [23] by relating instance lines to 0-ary *constants* from the given signature. A concrete binding is then given by the structure which interprets the LSC's signature; hence strictly speaking the quantification in the semantics definition is then ranging over structures. In addition to these constants for instance lines, we allow reference to a general set of constants called *specification variables* in the LSC which are also intended to be bound to concrete values.

Definition 8 (*LSC*). Let $B = (V, \Omega)$ be a signature and '*Msg*' a set of message names. A *Live Sequence Chart* $L = (\ell, ac, pch, m, X, am, quant)$ *over* B *and 'Msg'* consists of the following components:

- ℓ: The finite body of the LSC, comprising the following *body elements*: instance lines, synchronous and asynchronous message sending and reception, conditions, and local invariants.
- *ac*: The *activation condition*.
- *pch*: The (possibly empty) body of the *pre-chart*.
- *m*: The *annotation* of body elements as defined below.
- $X = \{x_1, \ldots, x_n\} \subseteq \Omega, n \in \mathbb{N}$: A finite set of 0-ary constants used as *logical variables*.
- $am \in \{initial, invariant\}$: The *activation mode*.
- $quant \in \{existential, universal\}$: The *(chart-)quantification*.

The bodies ℓ and '*pch*' of L together define the sets $insts(L)$ of instance lines, $sends(L)$ and $recvs(L)$ of *synchronous* and *asynchronous* message sendings and receptions respectively, and $conds(L) \supseteq \{ac\}$ of conditions and local invariants including the activation condition. Message sendings and receptions are required to be pairwise related, i.e. there exists a bijection μ between $sends(L)$ and $recvs(L)$,[3] and to be uniquely related to an instance line by ι.

The annotation m is a partial function which maps instance lines, messages, and conditions of L to an expression[4] obeying the following restrictions:

(i) If $p \in insts(L)$, then $m(p) = x \in X$, i.e. instance lines are only annotated by constants from X that are later subject to different bindings.

(ii) If $p \in sends(L) \cup recvs(L)$, then

$$m(p) = msg(expr_1, \ldots, expr_n) \in \bigcup_{k \in \mathbb{N}_0} Msg \times Expr(B)^k, n \in \mathbb{N}_0,$$

where $msg \in Msg$ and $expr_i \in Expr(B)$, i.e. the annotation of messages provides the name of the message and expressions that are typically interpreted as requirements on values (or parameters) carried by the message.

(iii) If $p \in conds(L)$, then $m(p) = expr \in Expr_{PL}(B)$ or

$$m(p) = \neg dest.msg(expr_1, \ldots, expr_n) \in \bigcup_{k \in \mathbb{N}_0} X \times Msg \times Expr(B)^k, n \in \mathbb{N}_0,$$

where $dest \in X$, $msg \in Msg$, and $expr_i \in Expr(B)$, i.e. conditions are basically predicate-logic expressions. The latter case is used to require the absence of messages in a local invariant.

(iv) If $p = ac$, then $m(p) = expr \in Expr_{PL}(B)$ or $m(p) =$

$$dest.msg(expr_1, \ldots, expr_n) \wedge expr \in \left(\bigcup_{k \in \mathbb{N}_0} X \times Msg \times Expr(B)^k \right) \times Expr(B),$$

[3] We will freely use μ to denote both μ and μ^{-1}; thus for each $p \in sends(L)$ or $p \in recvs(L)$, $\mu(p)$ is the related message reception or sending, respectively.

[4] In the sense of not formally introduced "LSC annotation expressions", i.e. not of one of the kinds introduced in Section 2.

$n \in \mathbb{N}_0$, where $dest \in X$, $msg \in Msg$, and $expr_i \in Expr(B)$, and $expr \in Expr_{PL}(B)$, i.e. the activation condition is basically an ordinary condition with additional means to activate on messages.

An LSC is called *unmapped* if m is not defined for any of the body elements and *partially mapped* if it is not defined for some of the body elements. □

The semantics of an LSC over signature $B = (V, \Omega)$ and message set Msg is explained *symbolically* by [22,24] in terms of a Timed Büchi Automaton (TBA). Using the annotation m and ι, the TBA can in our setting of LSCs without timing annotations be translated [37] into an LTL formula $\Phi(L)$ over $(V, \Omega \cup \{send_{msg}, receive_{msg} \mid msg \in Msg\})$ using constant function symbols $send_{msg}$ and $receive_{msg}$ which act as "placeholders" for the domain-dependent definition of message sending and reception.

By definition, the TBA, and hence the formula $\Phi(L)$, depends on the chart quantification *quant* of L: for an existential LSC, $quant = existential$, it expresses the *sequential composition* of the pre-chart and main chart, while for a universal LSC, $quant = universal$, it states that an observation of the pre-chart *implies* the main chart.

For example, consider the message arrow $(async\_snd_3, async\_rcv_3)$ in the LSC body in Fig. 2 between instance lines $inst_2$ and $inst_3$. An annotation

$$m(async\_snd_3) = m(async\_rcv_3) = msg(expr_1, \ldots, expr_n)$$

results in $send_{msg}(m(inst_2), m(inst_3), expr_1, \ldots, expr_n)$ and $receive_{msg}(m(inst_2), m(inst_3), expr_1, \ldots, expr_n)$, both occurring in $\Phi(L)$. The former is meant to be replaced by an expression which characterises the sending and the latter the reception of msg.

Note that synchronous and asynchronous messages are not distinguished on this level of *predicates*. The distinction is incorporated into the TBA and hence the LTL formula: for synchronous messages, sending and reception are required to occur *in the same* snapshot whereas for asynchronous messages, reception has to occur *at least* one snapshot later than sending.

When explaining LSCs for a particular application domain, it is often a matter of choice which of the domain's "observable events" are better mapped to synchronous and which to asynchronous messages of the LSC as we will see in Section 3.3 for the UML domain.

Definition 9 (*Compatibility Between Signatures and LSC and STS*). Let $B_1 = (V_1, \Omega_1)$ and $B_2 = (V_2, \Omega_2)$ be two signatures and X a finite set of 0-ary constants. We call B_1 *compatible with* B_2 *w.r.t.* X, $B_1 \subseteq_X B_2$, iff $V_1 \subseteq V_2$, $\Omega_1 \setminus X \subseteq \Omega_2$, and $\Omega_2 \cap X = \emptyset$.

Let $L = (\ell, ac, pch, m, X, am, quant)$ be an LSC over signature $B = (V, \Omega)$ and messages 'Msg'. Let $S = ((V, \Omega), \Theta, \rho)$ be an STS. The LSC L is called *compatible with* S iff B is compatible with (V, Ω) w.r.t. X. □

Definition 10 (*Satisfaction of an LSC*). Let $S = ((V, \Omega), \Theta, \rho)$ be an STS. Let $L = (\ell, ac, pch, m, X, am, quant)$, $X = \{x_1, \ldots, x_n\}$, $n \in \mathbb{N}$, be an LSC over signature $B = (V_B, \Omega_B)$ and messages 'Msg' compatible with S and let $\Phi(L)$ be the LTL formula representation of L.

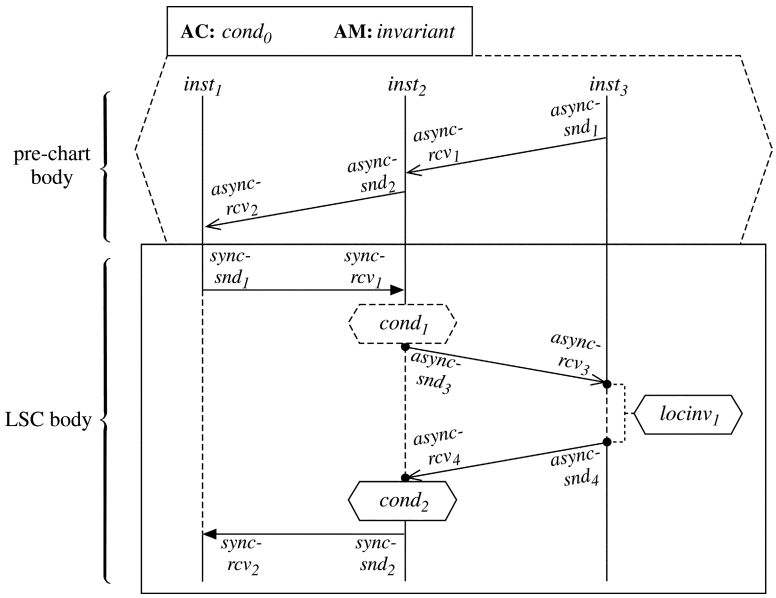


Fig. 2. **LSC example:** A graphical representation of the unmapped LSC
$L = (\{inst_1, inst_2, inst_3, sync\_snd_1, sync\_snd_2, sync\_rcv_1, sync\_rcv_2, async\_snd_3, async\_snd_4, async\_rcv_3, async\_rcv_4, cond_1, cond_2, locinv_1\}, cond_0, \{inst_1, inst_2, inst_3, async\_snd_1, async\_snd_2, async\_rcv_1 async\_rcv_2\}, \emptyset, X, invariant, universal)$,
i.e. with quantification *universal* as indicated by the solid line around the body of the LSC, activation condition $cond_0$, activation mode *invariant*, and non-empty pre-chart.
Note that the concrete graphical representation of LSCs generally follows MSCs, but here we use a concrete syntax more similar to that of UML Sequence Diagrams. The mandatory elements are, as usual for LSCs, depicted by solid lines and the possible elements by dashed lines.
Independent from the mapping, the LSC L is satisfied by all runs in which, any time after the two asynchronous messages in the pre-chart have been observed, a synchronous communication takes place between $inst_1$ and $inst_2$, and eventually—since the location between $sync\_rcv_1$ and $async\_snd_3$ is hot—an asynchronous communication takes place between $inst_2$ and $inst_3$. $cond_1$ is a cold condition (as indicated by the dashed border) in a simultaneous region with $async\_snd_3$; hence if at the *same* point in time when $async\_snd_3$ is observed, $cond_1$ *does not* hold, then the LSC is "exited successfully", i.e. the run satisfies the LSC.
Below $async\_rcv_3$, there is a *cold cut*; i.e. the current location on each instance line is cold and hence the following communication need not take place as long as the local invariant $locinv_1$ holds. $locinv_1$ is mandatory (as indicated by the solid line); thus if the condition $locinv_1$ is violated after $async\_rcv_3$ but before $async\_snd_4$, then the whole LSC is not satisfied.
The condition $cond_2$ is a mandatory condition, i.e. if $async\_rcv_4$ is observed and $cond_2$ does not hold at the same point in time, then the run does not satisfy L.
Since the subsequent locations are hot, both $sync\_snd_2$ and $sync\_rcv_2$ have to be observed in order to exit the LSC successfully.
Note that L may be activated multiple times in a run and even overlapping, for example if we had $m(async\_snd_1) = m(async\_snd_4)$. A run satisfies the LSC only if it is not violated in any activation. □

Let $\mathcal{M}_0 = (\mathcal{D}_0, \mathcal{I}_0)$ be a structure of (V, Ω) and $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ with $\mathcal{D} = \mathcal{D}_0 \cup \bigcup_{x \in X} \mathcal{D}_{type(x)}$, $\mathcal{I}|_\Omega = \mathcal{I}_0$, $\mathcal{I}$ defined on $\{send_{msg}, receive_{msg} \mid msg \in Msg\}$, and $\mathcal{I}$ not defined on X. The model S satisfies the LSC w.r.t. $\mathcal{M}$, $S \models_{\mathcal{M}} L$, iff

- *quant* = *existential* and
  - *am* = *initial* and
    $$\exists\, x_{0_1} \in \mathcal{D}_{type(x_1)}, \ldots, x_{0_n} \in \mathcal{D}_{type(x_n)} : S \models_{\mathcal{M}',\exists,0} ac \wedge \mathbf{X}\, \Phi(L), \text{ or}$$
  - *am* = *invariant* and
    $$\exists\, x_{0_1} \in \mathcal{D}_{type(x_1)}, \ldots, x_{0_n} \in \mathcal{D}_{type(x_n)} : S \models_{\mathcal{M}',\exists} ac \wedge \mathbf{X}\, \Phi(L), \text{ or}$$
- *quant* = *universal* and
  - *am* = *initial* and
    $$\forall\, x_{0_1} \in \mathcal{D}_{type(x_1)}, \ldots, x_{0_n} \in \mathcal{D}_{type(x_n)} : S \models_{\mathcal{M}',\forall,0} ac \implies \mathbf{X}\, \Phi(L), \text{ or}$$
  - *am* = *invariant* and
    $$\forall\, x_{0_1} \in \mathcal{D}_{type(x_1)}, \ldots, x_{0_n} \in \mathcal{D}_{type(x_n)} : S \models_{\mathcal{M}',\forall} ac \implies \mathbf{X}\, \Phi(L),$$

  where $\mathcal{M}' = (\mathcal{D}, \mathcal{I} \cup \{x_i \mapsto x_{0_i} \mid 1 \leq i \leq n\})$. □

3.3. LSCs for UML

In the following we refer to UML models in the definition of [7], i.e. a UML model is a tuple

$$M = (T, F, Sig, <, C, c_{root}, A),$$

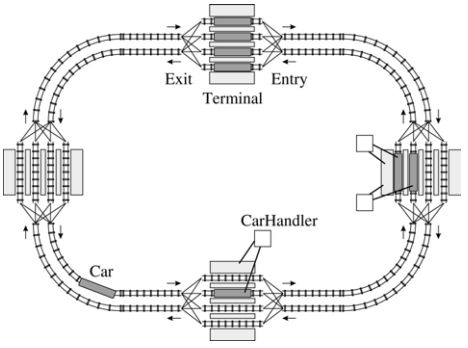
with T a set of basic types, F a set of predefined primitive functions, e.g. arithmetic operations on T, Sig a finite set of signals, $< \subsetneq Sig \times Sig$ a generalisation relation on signals, C a finite non-empty set of classes $c = (c.isActive, c.attr, c.ops, c.sm)$ comprising a predicate that indicates whether c is active (cf. [7]), a finite set of typed *attributes*, a finite set of *triggered operations*, and a *state-machine*. $c_{root} \in C$ is the class of the root object and $A \subseteq C$ the set of active classes called *actors*. The complete definition is provided by the companion paper [7]. In order to explain syntactical transformations on the transition predicate of $STS(M)$ in Section 4.3, in the following we assume F to contain $=: \tau \times \tau \to \mathbb{B}$, the comparison for equality on all types, and $(\cdot\ ?\ \cdot : \cdot) : \mathbb{B} \times \tau^2 \to \tau$, the if-then-else function.

We denote by T_c the *type of references to objects* of class $c \in C$ and by $T_C = \{T_c \mid c \in C\}$ the set of all T_c. For each class $c \in C$, O_c denotes the semantic type or domain of T_c, i.e. $\mathcal{D}_{T_c} = O_c$, and O_C the union of all O_c.

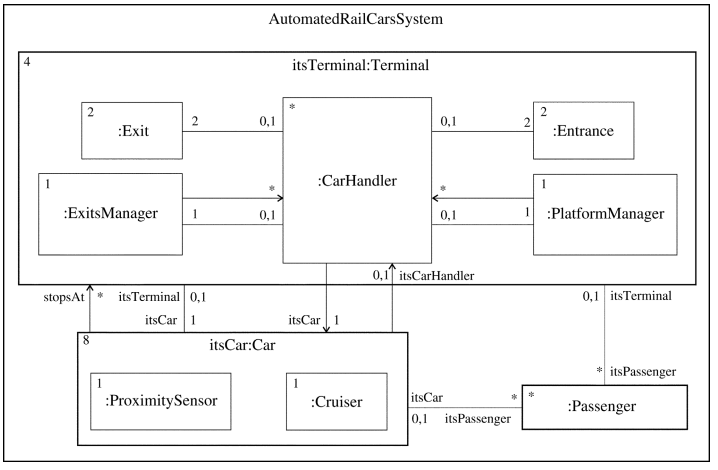
Example 11. Consider the LSC specification of the Automated Rail Cars System (*ARCS*) system [13] depicted in Fig. 3(a). The *ARCS* is a system of cars shuttling autonomously between a number of terminals. In order to avoid collisions there is a communication protocol for arrival and departure that cars and terminals or car handlers adhere to. Whenever a car approaches a terminal, the terminal creates a new car handler, which allocates and frees platforms and sets the switches, and passes its identity to the car. After a car has left the terminal, the car handler is destroyed. The behaviour of the classes is given in form of state-machines that we omit for brevity.[5]

Fig. 3(b) shows the class diagram of a model of the *ARCS*. Note that it is a particular instance of the system that is actually parametrised in the number of terminals, cars, and

[5] Also for brevity we do not elaborate further on the UML model of the *ARCS* here, but introduce the relevant concepts and classes as needed for our discussion; for details the reader is referred to the description in [13].



(a) Automated Rail Cars System.



(b) Class Diagram of the *ARCS*.

Fig. 3.

platforms per terminal. All system constituents except for so-called car handlers, which each manage a single arrival and departure procedure of a car at a terminal, are created at system initialisation time and not destroyed during a run of the system.

More formally, the *ARCS* model is represented as follows:

$$M = (T, F, \{\mathit{arrivReq}, \mathit{arrivAck}, \mathit{RIP}, \ldots\}, \emptyset, \{\mathit{Car} = (\mathit{true}, \{\mathit{stopsAt}, \mathit{itsTerminal}, \mathit{itsCarHandler}, \mathit{itsPassenger}, \mathit{itsProximitySensor}, \mathit{itsCruiser}, \ldots\}, \mathit{ops}, \mathit{sm}), \ldots\}, \mathit{AutomatedRailCarsSystem}, \emptyset),$$

showing only a subset of events and as an example parts of class *Car*. □

An LSC over M is basically an LSC over a signature derived from M with the set of events and triggered operations in M as set of messages Msg that satisfies a number of well-formedness rules:

Definition 12 (*LSC over UML Model*). Let $M = (T, F, Sig, <, C, c_{root}, A)$ be a UML model and $S = STS(M) = ((V, \Omega), \Theta, \rho)$ its semantics according to [7]. Let $L = (\ell, ac, pch, m, X, am, quant)$ be an LSC over $B = (V, \Omega)$ compatible with (V, Ω).

We call L an *LSC over M* iff the message set 'Msg' is

$$Msg = Sig \cup \{create_c \mid c \in C\} \cup \{destroy\} \cup \{reply_\tau \mid \tau \in T \cup T_C\} \cup \bigcup_{c \in C} c.ops$$

and L obeys the following well-formedness rules:

(i) **Instance Lines represent Objects:** If $p \in insts(L)$, then $m(p) \in X$ is of a reference type T_c for $c \in C$ and pairwise different from $m(p')$ for each $p' \in insts(L) \setminus \{p\}$, i.e. each instance line should be annotated by a different constant of a class type.

(ii) **Asynchronous Messages are Events:** If $p \in sends(L)$ is an asynchronous message sending or reception and $m(p) = msg(expr_1, \ldots, expr_n)$, then $msg = ev \in Sig$ and either $n = 0$ or n matches the number of parameters of ev and $type(expr_i)$ matches the type of the i-th parameter of ev.[6]

Note that the underlying semantics used does not distinguish identities of events; thus if an $ev \in Sig$ occurs more than once in L this may have "strange effects", for example a requirement of an event reception of ev can be fulfilled by receiving an event ev that has been sent long before L has been activated.

(iii) **Synchronous Messages are Operation Calls:** If $p \in sends(L)$ is a synchronous message sending or reception from instance line $i_1 \in insts(L)$ to $i_2 \in insts(L)$ and $m(p) = msg(expr_1, \ldots, expr_n)$, then $msg \in Msg \setminus Sig$ and m coincides on p and its related message, $\mu(p), m(\mu(p)) = m(p)$.

If $msg = op$ and $m(i_2)$ is of type T_c, $c \in C$, then $op \in c.ops$ and either $n = 0$ or n matches the number of parameters of op and $type(expr_i)$ matches the type of the i-th parameter of msg.

If $msg = reply_\tau$, then $n \leq 1$ and there is a uniquely identified synchronous message sending p' from i_2 to i_1, i.e. in the opposite direction, with $msg(p') \in c.ops$ for a $c \in C$ and $\tau = type_r(msg(p'))$. That is, a reply has to be related to an operation call.

If $msg = create_c$ or $destroy$, then $n = 0$ and p is the first or last, respectively, message or condition of the destination instance line and p is the only message annotated by creation or destruction respectively.

(iv) **Basic Conditions are Expressions:** If $p \in conds(L)$ is not the activation condition and not a local invariant, then $m(p) \in Expr_{PL}$.

[6] Providing only means to restrict either *all* or *none* of the parameters in the LSC is a matter of choice for brevity. The generalisation to restriction of only *some* parameters is straightforward if practical evaluation reveals a demand.

(v) **Messages in Conditions:** If $p \in conds(L)$ is the activation condition or a local invariant and $m(p) = dest.msg(expr_1, \ldots, expr_n)$ or $m(p) = \neg dest.msg(expr_1, \ldots, expr_n)$, then $dest$ is of type $T_c \in T_C$ and $msg \in c.ops \cup Sig$ and either $n = 0$ or n matches the number of parameters of msg and $type(expr_i)$ matches the type of the i-th parameter of msg.

(vi) **(Alive) Instances:** For each instance line $i \in insts(L)$ which does not contain a creation, the activation condition is conjoined with a term which requires i to be alive initially. For each instance line $i \in insts(L)$, the activation condition is conjoined with a term which requires i to be distinct from nil.

(vii) **Creation and Destruction:** For each creation $p \in sends(L) \cup recvs(L)$ from instance line $i_1 \in insts(L)$ to $i_2 \in insts(L)$ with $type(m(i_2)) = \tau$, $m(p) = create_\tau(i_1, i_2)$ and there exists a cold condition $q \in conds(L)$ at the topmost location of i_2 with $m(q) = m(i_2).create_\tau$ that allows us to legally exit the LSC in each run, where the creation operation does not create the object currently bound to i_2.

For each instance line $i \in insts(L)$ there is a local invariant which inhibits creation or destruction unless explicitly present in the LSC. For example, if $i \in insts(L)$ begins with a creation and ends with a destruction, the local invariant spans both locations (each exclusive). □

Note that $c.ops$ comprises only *triggered operations* of class $c \in C$, i.e. operations whose behaviour is defined by c's state-machine. So called *primitive operations* which are defined by a method are no longer visible on the semantical level of [7].

The requirements (vi) and (vii) "implement" our interpretation of creation and destruction, e.g. that an instance is supposed to live at least until its last location is reached which might be a destruction, and are included in Definition 12 for convenience. Practically they need not be represented in the concrete semantics but can be added automatically in a pre-processing step, invisible to the specifier.

As outlined in Section 3.2, we obtain an LTL formula $\Phi(L)$ for an LSC over a UML model which uses for example, for an asynchronous message sending from an instance i_1 to an instance i_2, the "placeholder" $send_{ev}(m(i_1), m(i_2), expr_1, \ldots, expr_n)$. To explain what it means for a UML model M to satisfy an LSC L, we use an extension of the STS semantics of M, $STS(M)$, according to [7]. The "placeholders" for the message send and receive are then interpreted as predicates over system variables including new system variables that are introduced to explicitly *observe* events and triggered operation based communications.

We need to introduce new system variables, since predicates over the unchanged model can only refer to the valuation of a *single* snapshot. But we want to observe e.g. the sending of an event $ev \in Sig$ from object o_1 to object o_2 in a snapshot r^{i+1} of a run $r \in runs(STS(M))$ *only if* the transition from r^i to r^{i+1} in $STS(M)$ corresponds to o_1 taking a transition which is annotated by an event sending action which enters an ev into the event queue of o_2's active object.

To observe the intended relation between two subsequent snapshots, we construct an *observer extension* of $STS(M)$ by introducing four kinds of new system variables $justsend_{ev}$, $justrecv_{ev}$ and $justcall_{op}$, $justret_{op}$ whose value has to be defined by an extended transition predicate s.t. for example $justsend$ becomes valid in snapshot r^{i+1} and holds the type and parameter values of the event that has been sent when taking the transition from

r^i to r^{i+1}. The first component of these variables is a boolean flag which indicates that the variable's value is valid; a single flag is sufficient due to the strictly interleaving and atomic nature of the underlying semantics. All of the variables carry sender, destination, and all parameters since e.g. the return value of a triggered operation is actually no longer visible in r^{i+1} in the *pending request table* [7].

Definition 13 (*Observer Extension*). Let $M = (T, F, Sig, <, C, c_{root}, A)$ be a UML model, and $S = STS(M) = (B, \Theta, \rho)$ its semantics according to [7].

The *observer extension* of S, $S_o = (B_o, \Theta_o, \rho_o)$, with $B_o = (V_o, \Omega_o)$ is obtained from S as follows:

(i) V is extended by the variables

$$justsend_{ev}, justrecv_{ev} : \mathbb{B} \times T_C \times T_C \times \bigcup_{ev \in Sig} type_{par}(ev), ev \in Sig,$$

to observe events, the variables

$$justcall_{op} : \mathbb{B} \times T_C \times T_C \times \bigcup_{\substack{c \in C \\ op \in c.ops}} type_{par}(op), op \in \bigcup_{c \in C} c.ops,$$

to observe triggered operation calls, and the variables

$$justret_{op} : \mathbb{B} \times T_C \times T_C \times \bigcup_{\substack{c \in C \\ op \in c.ops}} type_{r}(op), op \in \bigcup_{c \in C} c.ops,$$

to observe completion of triggered operations.

As introduced in [7], $type_{par}(ev)$ and $type_{par}(op)$ denote the Cartesian product of the event and operation parameter types respectively and $type_r(op)$ denotes the type of the return value of operation *op*.

(ii) Θ is changed s.t. the first four variables' first components get the value *false* initially.

(iii) $\rho_{\text{non\_op\_action}}$ which formalises taking a transition annotated with an action that is not an operation call[7] is conjoined with the following predicate:

$$(\gamma \equiv \text{“}r.send(ev, expr_1, \ldots, expr_n)\text{”} \wedge \neg sysfail' \implies justsend' := (true, ev, o, o.r, (expr_1, \ldots, expr_n)))$$

where γ denotes the transition considered and o the object taking the transition[8] (cf. [7] for the full set of abbreviations used).

Effectively, '*justsend*' observes the enqueueing of an event of type '*ev*' with destination '$o.r$' when object o takes a transition. It holds a valid value in the first snapshot where '*ev*' shows up in the queue.

[7] The transition predicate ρ is constructed using, among others, the sub-predicates $\rho_{\text{non\_op\_action}}$, $\rho_{\text{get\_event}}$, $\rho_{\text{discard\_event}}$, $\rho_{\text{init\_opcall\_or\_create}}$, and $\rho_{\text{pick\_up\_result}}$ that we refer to in this definition. For brevity, we do not reproduce the definition of ρ but refer the reader to the companion paper [7]. Furthermore, we use names like o and r that are present in the scope of these predicates (cf. [7]).

[8] The boolean system variable *sysfail* is used to indicate an undefined state of the system, e.g. in case a division by zero is attempted. We define the observer variables only for defined system states.

(iv) $\rho_{\text{get\_event}}$ and $\rho_{\text{discard\_event}}$ which formalise dispatching and discarding, respectively, of an event are conjoined with

$$(\neg sysfail' \implies justrecv' := (true, head(o.my\_ac.eq), \mathsf{nil}, o, o.ev_p{}'))$$

where nil is used as the sender, since the sender is not retained with the event, and o is the destination object. $head(o.my\_ac.eq)$ denotes the first entry in the event queue of o's active object and $o.ev_p{}'$ the attributes of o holding copies of the event parameters.

'*justrecv*' observes the dequeuing of an event of type '*ev*' with destination o. It holds in the first snapshot where '*ev*' has disappeared from the queue.

(v) $\rho_{\text{init\_opcall\_or\_create}}$ which formalises calling a triggered operation is conjoined with

$$(\neg sysfail' \implies justcall' := (true, prt'(o).op, o, r, prt'(o).params))$$

where o is the object initiating the call and r the destination.

'*justcall*' observes the operation call '*op*' when the caller changes status from executing to suspended and writes '*op*' with receiver r into its pending request table entry.

It holds in the first snapshot where o is suspended due to the call.

(vi) $\rho_{\text{pick\_up\_result}}$ which formalises picking up the result of a triggered operation call by the caller is conjoined with

$$(\neg sysfail' \implies justret' := (true, prt(o).op, o, prt(o).dest, prt'(o).result)).$$

'*justret*' observes return from operation call *op* when the caller o changes status from suspended back to executing or idle. It holds in the first snapshot where o is no longer suspended due to the call.

(vii) $\rho_{\text{non\_op\_action}}$ is in addition to (iii) conjoined with

$$(\gamma \equiv \text{“}destroy(expr)\text{”} \wedge \neg sysfail' \implies justcall' := (true, o, expr)).$$

Thus destruction, which is strictly speaking neither an event nor an operation call, is observed just like a triggered operation call.

(viii) ρ is finally changed s.t.

$$\begin{aligned}&(\neg sysfail' \implies\\ &\quad([\forall\, o \in O_C, o \neq \mathsf{nil} :\\ &\qquad(o.status = dormant \wedge o.status' = executing)\\ &\qquad\implies justsend_{create_c} = (true, o, o_1)))\end{aligned}$$

and s.t. for each other observer variable the first component gets the value '*false*' if the observer variable is not "assigned to" in a step. □

Note that in the above definition we chose to consider triggered operation calls as synchronous and observe only the call and the result being picking up, although they are actually *asynchronous* in [7] since a call can at the earliest be accepted one step after issuing the call. It is still to be assessed whether it is a better choice to consider triggered operation

calls to be represented by asynchronous messages in the LSC and whether it would be sensible to require all instance lines to be bound to *different* instances in the activation condition and the cold creation conditions.

The following definition finally provides the interpretation of the $send_{msg}$ and $receive_{msg}$ "placeholders" in terms of the observer extension and thereby defines the semantics of LSCs for UML.

Definition 14 (*Satisfaction of an LSC for UML*). Let $M = (T, F, Sig, <, C, c_{root}, A)$ be a UML model, and $S_o = (B_o, \Theta_o, \rho_o)$ the observer extension of its semantics. Let L be an LSC over M and $\mathcal{M}_0$ a structure of B_o.

The UML model M satisfies the LSC L w.r.t. $\mathcal{M}$, $M \models_{\mathcal{M}} L$, iff $STS(M) \models_{\mathcal{M}} L$ with $\mathcal{M}$ obtained as follows:

(i) For an event $ev \in Sig$ between instance lines $i_1, i_2 \in insts(L)$ and expressions $expr_1, \ldots, expr_N$, $N = 0$ or $N = n$, we set

$$\mathcal{M}[\![send_{ev}(i_1, i_2, \ldots)]\!](s) =_{df} \bigvee_{ev \leq \widehat{ev}} (\mathcal{M}_0[\![justsend_{\widehat{ev}}]\!](s) = (\mathcal{M}_0[\![true]\!](s), \mathcal{M}_0[\![m(i_1)]\!](s), \mathcal{M}_0[\![m(i_2)]\!](s), \ldots))$$

and

$$\mathcal{M}[\![receive_{ev}(i_1, i_2, \ldots)]\!](s) =_{df} \bigvee_{ev \leq \widehat{ev}} (\mathcal{M}_0[\![justrecv_{\widehat{ev}}]\!](s) = (\mathcal{M}_0[\![true]\!](s), \mathcal{M}_0[\![m(i_1)]\!](s), \mathcal{M}_0[\![m(i_2)]\!](s), \ldots)).$$

The ellipses are meant to abbreviate that if $send_{ev}$ and $receive_{ev}$ do not refer to expressions, then the parameter values of $justsend_{ev}$ and $justrecv_{ev}$ respectively are not considered. Otherwise the i-th parameter value of $justsend_{ev}$ and $justrecv_{ev}$ is to be compared with the i-th parameter expression of $send_{ev}$ and $receive_{ev}$ respectively. We use $ev_1 \leq ev_2$ as shorthand for $ev_1 = ev_2 \vee ev_1 < ev_2$.

(ii) For a triggered operation, creation, or destruction, $op \in c.ops \cup \{create_c, destroy\}$, $c \in C$, between instance lines $i_1, i_2 \in insts(L)$ we set

$$\mathcal{M}[\![send_{op}(i_1, i_2, \ldots)]\!](s) =_{df} \mathcal{M}[\![receive_{op}(i_1, i_2, \ldots)]\!](s) =_{df} (\mathcal{M}_0[\![justcall_{op}]\!](s) = (\mathcal{M}_0[\![true]\!](s), \mathcal{M}_0[\![m(i_1)]\!](s), \mathcal{M}_0[\![m(i_2)]\!](s), \ldots)).$$

Parameter expressions in $send_{op}$ or $receive_{op}$ are treated as explained above. Creation and destruction do not have parameters.

(iii) For a reply $op = reply_\tau$, $\tau \in T \cup T_C$ between instance lines $i_1, i_2 \in insts(L)$ we set

$$\mathcal{M}[\![send_{op}(i_1, i_2, \ldots)]\!](s) =_{df} \mathcal{M}[\![receive_{op}(i_1, i_2, \ldots)]\!](s) =_{df} (\mathcal{M}_0[\![justret_{op}]\!](s) = (\mathcal{M}_0[\![true]\!](s), \mathcal{M}_0[\![m(i_1)]\!](s), \mathcal{M}_0[\![m(i_2)]\!](s), \ldots)).$$

The optional parameter expression in $send_{op}$ or $receive_{op}$ is treated as explained above. □

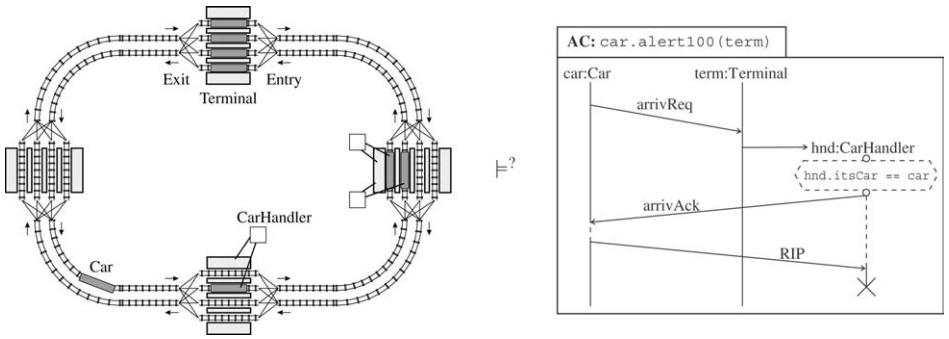


Fig. 4. **LSC over *ARCS*:** The *LSC* states the following requirement on the arrival and departure procedure on a very high level of abstraction: The LSC is activated whenever *car* (an instance of class *Car*) is 100 units ahead of a *Terminal term* which is indicated by the *car* receiving an *alert100* event announcing the approached terminal. The *car* eventually initiates the entering protocol by sending an *arrivReq* event to *term* whose identity it obtained from one of its sensors (not shown in the LSC).
The terminal eventually creates an instance of a class *CarHandler* that subsequently manages the whole entering and leaving procedure, i.e. it reserves and frees platforms and exits within *term* and it communicates with the switches.
Once the car handler has obtained a platform and set the switch, it sends an *arrivAck* event back to the car which then enters the terminal. The *arrivAck* event carries the address of the *CarHandler*. The *car* actually does not communicate with the terminal directly after sending the *arrivReq* event.
If the *arrivAck* event has been observed, the current locations on all instance lines are cold (as indicated by the dashed lines; the middle instance line is implicitly cold after its last message); thus it is not required that the *car* eventually leaves the terminal. When the car is about to leave the terminal, it sends another request to its car handler which sends back a granting event once the switches of the desired exit are set and free (not shown in the LSC).
After having left the terminal, *car* sends an event *RIP* to its *CarHandler* which causes *hnd* to release the reservations of platform and exit and finally to destroy itself. □

4. Infinite state system verification

Consider the LSC specification for the *ARCS* as introduced in Section 3.3 depicted in Fig. 4. Via the LSC semantics of Section 3, it can be checked whether the system satisfies the specification by checking *all* concrete bindings of *Car* and *Terminal* objects to instance lines. But intuitively, it should be sufficient to check *a single* concrete binding for e.g. the *Car* identity car_0 since if the instance with this identity *always* behaves as required, then *every* *Car* behaves like that, since they are all instances of the same class with the same behaviour. The deeper reason is that new objects are chosen non-deterministically at creation time; thus if an object car_1 would violate the specification in a run of the system, then there existed a run which chooses car_0 instead of car_1 at creation time and thus there existed a run where car_0 violates the property, too. Thus we want to exploit *symmetries* of the transition system induced by, e.g., all car instances being of the same class, and prove only a finite set of representative cases which imply all other cases.

Yet reducing the whole task to a finite number of concretely bound representative cases does not reduce the model at all; the state space in the semantics of [7] still provides entries for an unbounded number of objects.

To reach the domain of standard finite state verification methods, in a second step we try to prove each representative task on a finite over-approximation of the original model in which only as many objects are concretely represented as needed for the proof, where the sufficient number is obtained by iteratively refining the abstraction. The over-approximation is obtained by introducing a new object identity $\perp$ that represents the identity of "any other object different from all concretely represented ones" and by changing the system description s.t. dereferencing $\perp$ yields "guessed" values which are a superset of all possibly observable outcomes in the original system.

In the following Sections 4.2 and 4.3, we provide a formal basis for the just outlined direction in full generality, which is basically a transfer to the UML domain of parts of the methodology presented in [32] for the verification of a parametrised processor design with replicated components which in turn is based on the work of [17], yet we provide all proofs for completeness. But we begin this section with a brief discussion of the more common way to exploit system symmetries.

4.1. The other kind of symmetry reduction: the quotient model

The idea to exploit in formal verification the symmetries of a system caused by replicated components, like processors in a cache-coherency protocol, actually dates back to 1993. Emerson and Sistla [10] and Ip and Dill [17] independently discovered that symmetries of transition systems can be exploited to prove certain properties on the *quotient graph* by the equivalence relation induced by symmetry instead of on the full transition system.

The authors of [17] even provided criteria that allow one to declare and syntactically check symmetric (or *scalarset*) data-types in the system description language of the Murφ model-checker [18].

The disadvantage of all quotient graph approaches is that the set of properties is restricted to safety properties which are independent from individual identities, for example, "none of the symmetric components of the system runs into a deadlock" [17] or to LTL properties which are themselves symmetric, that is invariant under permutation of indices [10]. The quotient graph approach to symmetry based model reduction is in general not applicable for LSC verification since on the one hand LSCs are in general not just safety properties and on the other hand LSCs are in general meant to distinguish different instances of a class. In the quotient approach of [17,10], instances are no longer distinguishable.

A recent example for an application of the quotient approach is the dSPIN [16] variant of the SPIN model-checker. It exploits heap symmetries in software verification; i.e. system states which differ only in the allocation of objects into memory places on the heap are equivalent on the program level in languages like Java, and analogously process (allocation) symmetries [15]. The published results as yet comprise only checking for absence of deadlock.

4.2. Query reduction

A different way to exploit symmetry—for which the authors of [39] coined the term "Query Reduction"—was first demonstrated by the author of [29,31,32] for general

temporal logic properties of the form of a universal quantification over a *symmetric or scalarset type*. In this case it is sufficient to prove only a finite representative set of concrete bindings since all other bindings and hence the whole quantification are implied by symmetry.

Query reduction applies to all systems where the state of replicated components is kept in an array data structure indexed by a symmetric type and to properties which claim that for all indices i in the array data structure a property $\phi_0(i)$ holds, and thus in particular to our interpretation of LSCs over UML models.

When proving the representative cases separately, there is not only an anticipated benefit from the smaller size of the formulae compared to the original quantification. The representative formulae are also more specialised than the original ones in that they refer to only a concrete binding of the quantified variables, and thus standard model reduction techniques like cone-of-influence reduction [1] can be applied more effectively. Thus query reduction is strictly speaking *not at all* a *model reduction*, but it is only a decomposition of formulae s.t. standard model reduction techniques yield better results.

But cone-of-influence reduction alone does not address the problem that the state space of a UML model is in general infinite if there are no finite bounds on the number of objects alive in each state provided. Therefore we propose to apply, in a second step, the abstract interpretation Data-type Reduction [32] which represents only finitely many objects concretely and considers an over-approximation of the behaviour of all other objects.

Note that the length of the event queue is a priori still unbounded in a UML design; thus for the scope of this paper we assume a finite upper bound on the length of event queues to reach the domain of automatic techniques for finite state verification. For the category of so-called mode separated models [5] presents an exact abstraction that eliminates queues from models comprising communicating state charts and yields a finite representation.

The remainder of this section is structured as follows. In Section 4.2.1 we briefly provide the concept of automorphisms and define scalarsets in a more general way s.t. a permutation on the domain of the scalarset induces an automorphism of the transition system. This allows us to prove that certain LTL formulae over 0-ary constants of scalarset type are invariant under permutation of the constants' values. Section 4.2.2 introduces the notion of a representative set and claims the existence of a finite representative set for each of the LTL properties considered in Section 4.2.1. Finally Section 4.2.3 demonstrates LSCs over the STS semantics of UML [7] in the interpretation of Section 3 as a prominent application domain for the results of the previous sections.

4.2.1. Permutations and automorphisms

Definition 15 (*Permutation and Automorphism*). Let A be a countably infinite set. We call a bijection $\pi : A \to A$ which coincides with the identity on A on all but finitely many elements a *permutation on* A. The set of all permutations on A is called $Sym(A)$ and we use $Sym_a(A) \subsetneq Sym(A)$ to denote the set of all permutations on A that coincide on $a \subseteq A$ with the identity.

Let $S = (B, \Theta, \rho)$ be an STS and $\mathcal{M}$ a structure of B.

A permutation $\pi \in Sym(\Sigma_B)$ is called an *automorphism of S w.r.t.* $\mathcal{M}$ iff

(i) $\forall\, s \in \Sigma_B : \mathcal{M}[\![\Theta]\!](s) \implies \mathcal{M}[\![\Theta]\!](\pi(s))$,

(ii) $\forall\, s, s' \in \Sigma_B : \mathcal{M}[\![\rho]\!](s, s') \implies \mathcal{M}[\![\rho]\!](\pi(s), \pi(s'))$. □

A permutation on the domain of an unstructured type induces a permutation on the domains of record and array types as follows.

Definition 16 (*Induced Permutation*). Let $B = (V, \Omega)$ be a signature and $\mathcal{M}$ a structure of B. Let $\tau \in T_B$ be an unstructured type with at most one special element $\mathsf{nil} \in \mathcal{D}_\tau$. We use $\mathsf{nil}(\tau) \subseteq \mathcal{D}_\tau$ to denote the (possibly) empty set of special elements in the domain of τ. Let $\pi \in Sym_{\mathsf{nil}(\tau)}(\mathcal{D}_\tau)$ be a permutation on the domain of τ with $\pi(\mathsf{nil}) = \mathsf{nil}$ if τ has a special element. We call the permutation $\pi_\mathcal{D}$ on all types used in B defined pointwise for values $x \in \mathcal{D}$ by

$$\pi_\mathcal{D}(x) =_{df} \begin{cases} \pi(x) & , \text{if } x \in \mathcal{D}_\tau \\ \{i \mapsto \pi_\mathcal{D}(x(\pi_\mathcal{D}^{-1}(i))) \mid i \in \tau\} & , \text{if } x \in \mathcal{D}_{\tau \to \tau'} \\ (\pi_\mathcal{D}(x_1), \ldots, \pi_\mathcal{D}(x_n)) & , \text{if } x \in \mathcal{D}_{\tau_1 \times \cdots \times \tau_n} \\ x & , \text{otherwise} \end{cases}$$

the *induced permutation of* π *on* $\mathcal{D}$. □

The following definition provides the central notion of the current section, a kind of symmetrical data-type called scalarset on which all constants are consistent or invariant under permutation on the domains of their operands, i.e. independent from particular values of the type. An example for invariance is the comparison for equality whose truth-value does not change if a permutation is applied to both of its operands. An example for consistency is the if-then-else operator for the type: it always yields the first or second value depending on the conditional expression. In this sense, the outcome does not depend on the values of the second and third parameters.

Definition 17 (*Scalarset*). Let $S = (B, \Theta, \rho)$ be an STS and $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ a structure of B. Let $\tau_s \in T_B$ be an unstructured type with $|\mathsf{nil}(\tau_s)| \leq 1$. τ_s is called a *scalarset type (w.r.t.* $\mathcal{M}$*)* iff

$$\forall\, \pi \in Sym_{\mathsf{nil}(\tau_s)}(\mathcal{D}_{\tau_s})\ \forall\, f : \tau_1 \times \cdots \times \tau_n \to \tau \in \Omega\ \forall\, x_1 \in \mathcal{D}_{\tau_1}, \ldots x_n \in \mathcal{D}_{\tau_n} :$$
$$\mathcal{I}(f)(x_1, \ldots, x_n) = \pi_\mathcal{D}^{-1}\left(\mathcal{I}(f)(\pi_\mathcal{D}(x_1), \ldots, \pi_\mathcal{D}(x_n))\right),$$

i.e. for each permutation π, each constant f is invariant under $\pi_\mathcal{D}$ for $\tau \neq \tau_s$ and consistent with $\pi_\mathcal{D}$ for $\tau = \tau_s$. □

Note that the check for legal use[9] of a scalarset type in a system description is a simple type-checking task, as first developed by [17].

The following lemma claims that from the use of a scalarset type in a system description we can infer symmetries (or automorphisms) of the transition relation which are directly related to permutations on the scalarset type.

[9] An example for illegal use is as simple as an occurrence of an expression of scalarset type as operand of an addition.

Lemma 18 (*System Symmetry*). *Let $S = (B, \Theta, \rho)$ be an STS and $\mathcal{M}$ a structure of B. Let τ_s be a scalarset type w.r.t. $\mathcal{M}$. Then the* induced permutation $\tilde{\pi}$ of π on Σ_B *defined pointwise on $s \in \Sigma_B$ for variables $v \in V$ by $\tilde{\pi}(s)(v) =_{df} \pi_{\mathcal{D}}(s(v))$ is an automorphism of S w.r.t. $\mathcal{M}$.* □

Proof. Analogous to that of [17]. □

Intuitively, if two states $s, s' \in \Sigma$ are related since they satisfy the transition relation, then all pairs of states $\tilde{\pi}(s), \tilde{\pi}(s') \in \Sigma$ reachable by a permutation π are related, too, since the transition relation is invariant under permutation. If a state $s_0 \in \Sigma$ is initial, then all states $\tilde{\pi}(s_0) \in \Sigma$ reachable by a permutation $\tilde{\pi}$ are initial, too, since the initiation relation is also invariant under permutation.

The work [17] gives type-checking rules for the particular system description language of their Murφ model-checker, and calls those types which adhere to the rules scalarset types. These criteria are sufficient but not necessary for scalarset types in our more general definition. But they do in particular allow the modeller to explicitly declare scalarset types for attributes of types other than object references in the model that can be type-checked and then also be exploited in formal verification.

The following lemma employs the criteria of [17] to detect symmetries in a system description in which a priori no scalarset types are declared.

Lemma 19 (*Scalarset [17]*). *Let $S = (B, \Theta, \rho)$ be an STS and $\tau \in T_B$ with $|\mathsf{nil}(\tau)| \leq 1$. Let $\mathcal{M}$ be a structure of B which assigns the comparison operator '$=: \tau \times \tau \to \mathbb{B}$' the natural semantics. Then the type τ can be replaced by a scalarset type if the predicates Θ and ρ obey the following syntactical rules:*

S1 *Values of τ are not used literally, i.e. there are no 0-ary constants except for the special element* nil.

S2 *Terms of type τ may be compared for equality. In a comparison, both sides must be terms of exactly the same scalarset type.*

S3 *If the left hand side of an "assignment" is of type τ, then the right hand side must be of exactly the same type.*

S4 *If τ is the index type of some dimensions of an array a, then a is only indexed by terms of type τ for this dimension.*

S5 *A variable of scalarset type τ_s may be existentially quantified or used as the running index of a for-loop if the body of the loop is independent from the order of the iterations. A sufficient (but not necessary) restriction for obtaining this property is that the set of variables written within each iteration are disjoint from the set of variables referenced (read or written) during all other iterations.*

S6 *Other operations are not allowed, i.e. no other constants are applied to expressions of type τ. In particular, expressions of scalarset types may not be used as operands of '+' or "cast" into an integer type.* □

Proof. The criteria restrict the use of expressions of type τ to cases which satisfy the restrictions of Definition 17, thus τ can be replaced by a fresh type τ_s with the same domain as τ for which only the comparison operator '=' and possibly nil are defined. □

Note that we do not cover loops with a variable of scalarset type as running index or first-order predicates over a variable of scalarset type in our Definition 17 for brevity. They can easily be integrated into the theory of query reduction assuming (S5). For the case of UML models, some special treatment is needed to ensure (S5), since a loop over the type of an association index typically references the same variables in all iterations when the intention of the loop is for example to send events to all associated objects (cf. Section 4.2.3 and Section 4.3).

The following lemma claims that an LTL property over a set of constants of a scalarset type τ_s is invariant under renaming of the constants (which we formalise as re-interpretation by the structure to match our interpretation of LSCs from Section 3), or, in other words, that proving a single binding of the constants implies all bindings which are reachable by a permutation on the scalarset type's domain.

Lemma 20 (*Property Symmetry*). *Let* $S = ((V_0, \Omega_0), \Theta, \rho)$ *be an STS. Let* $\mathcal{M}_0$ *be a structure of* (V_0, Ω_0) *and* $\tau_s \in T_{B_0}$ *a scalarset type w.r.t.* $\mathcal{M}_0$. *Let* X *be a finite set of 0-ary constants of type* τ_s *and* ϕ *an LTL formula over* $B = (V, \Omega)$ *with* $X \subseteq \Omega$ *s.t.* $B \subseteq_X (V_0, \Omega_0)$. *Let* $\mathcal{M}$ *be a structure of* B.

Then for each permuted structure

$$\mathcal{M}^\pi =_{df} (\mathcal{D}, \mathcal{I}|_{\Omega\setminus X} \cup \{x \mapsto \pi(\mathcal{I}(x)) \mid x \in X\})$$

of B *induced by* $\pi \in Sym_{\mathsf{nil}(\tau_s)}(\mathcal{D}_{\tau_s})$ *and for each* $q \in \{\exists; \exists, 0; \forall; \forall, 0\}$ *we have*

$$S \models_{\mathcal{M},q} \phi \iff S \models_{\mathcal{M}^\pi,q} \phi. \quad \square$$

Proof. It is sufficient to prove the direction ' $\Longrightarrow$ '. The other direction follows by symmetry.

Let $q = \exists$ and $S \models_{\mathcal{M},\exists} \phi$, i.e. $\exists\, r \in runs(S)\ \exists\, i \in \mathbb{N} : r/i \models_{\mathcal{M},\exists} \phi$. Let $\mathcal{M}^\pi$ be a structure permuted by π as defined above. Choose $r_\pi =_{df} \tilde{\pi}(r^0)\,\tilde{\pi}(r^1)\,\tilde{\pi}(r^2)\ldots$ which is in $runs(S)$ since $\tilde{\pi}$ is an automorphism. Then $r_\pi/i \models_{\mathcal{M}^\pi,\exists} \phi$, and hence $S \models_{\mathcal{M}^\pi,\exists} \phi$, follows by induction over the structure of ϕ:

- $\phi \equiv expr$: Since $expr \in Expr_{PL}(B)$ and τ_s is not boolean, it is sufficient to consider the following predicates of $expr$ as an induction basis:
  - $expr \equiv f(expr_1, \ldots, expr_n)$ where f is a predicate and each $expr_i$ of a type τ is a variable $v_i \in V$ or a constant $x_i \in X$ or an array index expression $a_i[v_i]$ or $a_i[x_i]$ or a selection expression $p_i =_{df} expr_i.c_i$, i.e. a projection onto a component c_i of a record type. Then

$$\begin{aligned}
&\mathcal{M}^\pi[\![f(expr_1, \ldots, expr_n)]\!](r_\pi^i) \\
&\quad = \mathcal{I}'(f)(\mathcal{M}^\pi[\![expr_1]\!](r_\pi^i), \ldots, \mathcal{M}^\pi[\![expr_n]\!](r_\pi^i)) \\
&\quad = \mathcal{I}'(f)\begin{pmatrix} \mathcal{M}^\pi[\![v_1]\!](r_\pi^i) & & \mathcal{M}^\pi[\![v_n]\!](r_\pi^i) \\ \mathcal{M}^\pi[\![x_1]\!](r_\pi^i) & & \mathcal{M}^\pi[\![x_n]\!](r_\pi^i) \\ \mathcal{M}^\pi[\![a_1[v_1]]\!](r_\pi^i), & \ldots, & \mathcal{M}^\pi[\![a_n[v_n]]\!](r_\pi^i) \\ \mathcal{M}^\pi[\![a_1[x_1]]\!](r_\pi^i) & & \mathcal{M}^\pi[\![a_n[x_n]]\!](r_\pi^i) \\ \mathcal{M}^\pi[\![p_1]\!](r_\pi^i) & & \mathcal{M}^\pi[\![p_n]\!](r_\pi^i) \end{pmatrix}
\end{aligned}$$

$$= \mathcal{I}'(f)\begin{pmatrix} r^i_\pi(v_1) & r^i_\pi(v_n) \\ \mathcal{I}'(x_1) & \mathcal{I}'(x_n) \\ r^i_\pi(a_1)(r^i_\pi(v_1)) & , \ldots , \; r^i_\pi(a_n)(r^i_\pi(v_n)) \\ r^i_\pi(a_1)(\mathcal{I}'(x_1)) & r^i_\pi(a_n)(\mathcal{I}'(x_n)) \\ \mathcal{M}^\pi[\![p_1]\!](r^i_\pi) & \mathcal{M}^\pi[\![p_n]\!](r^i_\pi) \end{pmatrix}$$

$$\underset{\pi_\mathcal{D}|_{\tau_s}=\pi}{=} \mathcal{I}'(f)\begin{pmatrix} \pi_\mathcal{D}(r^i(v_1)) & \pi_\mathcal{D}(r^i(v_n)) \\ \pi_\mathcal{D}(\mathcal{I}(x_1)) & \pi_\mathcal{D}(\mathcal{I}(x_n)) \\ \pi_\mathcal{D}(r^i_\pi(a_1)(\pi_\mathcal{D}^{-1}(\pi_\mathcal{D}(r^i(v_1))))) & , \ldots , \; \pi_\mathcal{D}(r^i_\pi(a_n)(\pi_\mathcal{D}^{-1}(\pi_\mathcal{D}(r^i(v_n))))) \\ \pi_\mathcal{D}(r^i_\pi(a_1)(\pi_\mathcal{D}^{-1}(\pi_\mathcal{D}(\mathcal{I}(x_1))))) & \pi_\mathcal{D}(r^i_\pi(a_n)(\pi_\mathcal{D}^{-1}(\pi_\mathcal{D}(\mathcal{I}(x_n))))) \\ \pi_\mathcal{D}(\mathcal{M}[\![p_1]\!](r^i)) & \pi_\mathcal{D}(\mathcal{M}[\![p_n]\!](r^i)) \end{pmatrix}$$

$$\underset{\substack{\text{Lemma 18,}\\ \mathcal{I}'(f)=\mathcal{I}(f)}}{=} \mathcal{I}(f)\begin{pmatrix} r^i(v_1) & r^i(v_n) \\ \mathcal{I}(x_1) & \mathcal{I}(x_n) \\ r^i(a_1)(r^i(v_1)) & , \ldots , \; r^i(a_n)(r^i(v_n)) \\ r^i(a_1)(\mathcal{I}(x_1)) & r^i(a_n)(\mathcal{I}(x_n)) \\ \mathcal{M}[\![p_1]\!](r^i) & \mathcal{M}[\![p_n]\!](r^i) \end{pmatrix}$$

$$= \quad \mathcal{M}[\![f(expr_1, \ldots, expr_n)]\!](r^i) = true,$$

and hence $r_\pi / i \models_{\mathcal{M}^\pi} expr$.

– $expr \equiv a[expr]$ where a is of type $\tau_s \to \mathbb{B}$ and $expr$ is a constant $x \in X$ or a variable $v \in V$ of type τ_s or an array index expression $b[v]$ or $b[x]$ or a selection expression $p =_{df} expr_0.c$, i.e. a projection onto a component c of type τ_s of a record type. Then

$$\mathcal{M}^\pi[\![a[expr]]\!](r^i_\pi) = r^i_\pi(a)(r^i_\pi(expr))$$

$$= \pi_\mathcal{D}\left(r^i(a)\left(\pi_\mathcal{D}^{-1}\begin{pmatrix} r^i_\pi(v) \\ \mathcal{I}'(x) \\ r^i_\pi(b)(r^i_\pi(v)) \\ r^i_\pi(b)(\mathcal{I}'(x)) \\ \mathcal{M}^\pi[\![p]\!](r^i_\pi) \end{pmatrix}\right)\right)$$

$$\underset{\pi_\mathcal{D}|_{\tau_s}=\pi}{=} \pi_\mathcal{D}\left(r^i(a)\left(\pi_\mathcal{D}^{-1}\begin{pmatrix} \pi_\mathcal{D}(r^i(v)) \\ \pi_\mathcal{D}(\mathcal{I}(x)) \\ \pi_\mathcal{D}(r^i_\pi(b)(\pi_\mathcal{D}^{-1}(\pi_\mathcal{D}(r^i(v)))))) \\ \pi_\mathcal{D}(r^i_\pi(b)(\pi_\mathcal{D}^{-1}(\pi_\mathcal{D}(\mathcal{I}(x))))) \\ \pi_\mathcal{D}(\mathcal{M}[\![p]\!](r^i)) \end{pmatrix}\right)\right)$$

$$= \quad \pi_\mathcal{D}\Bigg(\underbrace{r^i(a)\begin{pmatrix} r^i(v) \\ \mathcal{I}(x) \\ r^i(b)(r^i(v)) \\ r^i(b)(\mathcal{I}(x)) \\ \mathcal{M}[\![p]\!](r^i) \end{pmatrix}}_{\in \mathcal{D}_\mathbb{B}}\Bigg)$$

$$= \mathcal{M}[\![a[expr]]\!](r^i) = true,$$

and hence $r_\pi / i \models_{\mathcal{M}^\pi} expr$.

The general case $r_\pi / i \models_{\mathcal{M}^\pi} expr$ follows by induction (using the consistency property of Definition 17), and hence $r_\pi \models_{\mathcal{M}^\pi,\exists} \phi$.

- $\phi \equiv f \vee g$: Then $r \models_{\mathcal{M},\exists} f$ or $r \models_{\mathcal{M},\exists} g$.
  Thus by the induction hypothesis $r_\pi \models_{\mathcal{M}^\pi,\exists} f$ or $r_\pi \models_{\mathcal{M}^\pi,\exists} g$.
- $\phi \equiv \neg f$: analogous to the previous case.
- $\phi \equiv \mathbf{X} f$: Then $r/i + 1 \models_{\mathcal{M},\exists} f$.

Thus $r_\pi/i+1 \models_{\mathcal{M}^\pi,\exists} f$ by the induction hypothesis.

- $\phi \equiv \mathbf{G}\, f$: Then for all $j \geq i$, $r/j \models_{\mathcal{M},\exists} f$.
  Thus also $r_\pi/j \models_{\mathcal{M}^\pi,\exists} f$ by the induction hypothesis.
- $\phi \equiv f\, \mathbf{U}\, g$: Then there exists $k \geq i$ s.t. $r/k \models_{\mathcal{M},\exists} g$ and for all $i \leq j < k$, $r/j \models_{\mathcal{M}^\pi,\exists} f$. Thus also $r_\pi/k \models_{\mathcal{M}^\pi,\exists} g$ and for all $i \leq j < k$, $r_\pi/j \models_{\mathcal{M}^\pi,\exists} f$, by the induction hypothesis.

The case $q = \exists, 0$ is obtained analogously, the cases $q = \forall$ and $q = \forall, 0$ similarly by contradiction. □

4.2.2. Query reduction

In order to apply Lemma 20, we are interested in a set of representative bindings which imply as many other cases as possible. The following definition introduces the notion of a representative set and Lemma 22 ensures the existence of a finite representative set for each property over finitely many scalarset constants.

The representative bindings are selected in order to cover all cases of equalities within the set of affected constants. For example, if there are two quantification constants x_1, x_2 in the property, we need at least two concrete bindings $\mathcal{M}'_1, \mathcal{M}'_2$: one which represents all bindings which bind the same value to both constants, $\mathcal{M}'_1[\![x_1]\!] = \mathcal{M}'_2[\![x_2]\!]$, and one which represents the bindings with different values, $\mathcal{M}'_1[\![x_1]\!] \neq \mathcal{M}'_2[\![x_2]\!]$.

Definition 21 (*Representative Set*). Let τ_s be a scalarset type. A set $R \subseteq (\mathcal{D}_{\tau_s} \setminus \mathsf{nil}(\tau_s))^n$, $n \in \mathbb{N}$, is called a *representative set for* τ_s^n iff

$$\forall\, (x_1, \ldots, x_n) \in \mathcal{D}^n_{\tau_s}\ \exists\, r = (r_1, \ldots, r_n) \in R, \pi \in \mathit{Sym}(\tau_s) : \\ (\pi(r_1), \ldots, \pi(r_n)) = (x_1, \ldots, x_n).$$

R is called a *minimal representative set* if all proper subsets $R' \subsetneq R$ are not representative. □

Lemma 22 (*Representative Set*). *Let τ_s be a scalarset type and $n \geq 1$. Then there exists a finite representative set R for τ_s^n.* □

Proof. Choose $r_1, \ldots, r_n \in \mathcal{D}_{\tau_s} \setminus \mathsf{nil}(\tau_s)$ pairwise different.
Set $R = \{r_1\} \times \{r_1, r_2\} \times \cdots \times \{r_1, \ldots, r_n\}$. □

Note that the R constructed in Lemma 22 is in general not minimal, since e.g. in the case of $n = 3$ the tuples (r_1, r_1, r_2) and (r_1, r_1, r_3) are equivalent but both are in R.

The following corollary puts it all together and re-states Lemma 20 for the form of quantified LTL formulae we use for LSCs.

Corollary 23 (*Query Reduction*). *Let (V_0, Ω_0) be a signature and $S = ((V_0, \Omega_0), \Theta, \rho)$ be an STS. Let $\mathcal{M}_0$ be a structure of B_0 and $\tau_s \in T_{B_0}$ a scalarset type w.r.t. $\mathcal{M}_0$. Let X be a finite set of 0-ary constants of type τ_s and ϕ an LTL formula over $B = (V, \Omega)$ with $X \subseteq \Omega$ s.t. $B \subseteq_X B_0$.*

Denote by $\{o_1, \ldots, o_n\} \subseteq X$ *the constants of type* τ_s *and by* $\{x_1, \ldots, x_m\} = X \setminus \{o_1, \ldots, o_n\}$ *the constants not of type* τ_s. *Let* R *be a representative set for* τ_s^n. *Then*

$$(\forall\, (o_{0_1}, \ldots, o_{0_n}) \in R \quad \forall\, x_{0_1} \in \mathcal{D}_{type(x_1)}, \ldots, x_{0_m} \in \mathcal{D}_{type(x_m)} : S \models_{\mathcal{M}',q} \phi) \tag{1}$$

$$\iff (\forall\, o_{0_1}, \ldots, o_{0_n} \in \mathcal{D}_{\tau_s} \quad \forall\, x_{0_1} \in \mathcal{D}_{type(x_1)}, \ldots, x_{0_m} \in \mathcal{D}_{type(x_m)} : S \models_{\mathcal{M}'',q} \phi) \tag{2}$$

for $q \in \{\exists; \exists, 0; \forall; \forall, 0\}$ *and* $\mathcal{M}'$, $\mathcal{M}''$ *constructed as in Definition* 14. □

Now given a verification task ϕ over a signature of an STS and finitely many 0-ary constants used exclusively in ϕ we obtain for each scalarset type τ_s a finite representative set R by Lemma 22 and can apply Corollary 23 iteratively for each scalarset type and thereby obtain a *finite* number of verification tasks if all 0-ary constants are of a scalarset type.

4.2.3. Verifying LSCs against UML models

The verification of LSCs against UML models in the semantics of [7] provides a prominent application of the theory of query reduction as presented in the preceding sections since on the one hand the semantics of an LSC is of the form of an LTL expression over the transition system's signature and an additional set of 0-ary constants and on the other hand the object reference types T_c in [7] are in fact scalarset types:

Lemma 24 (*Scalarsets in UML*). *Let* $M = (T, F, Sig, <, C, c_{root}, A)$ *be a UML model and STS*(M) *its semantics according to [7].*

(i) *All object reference types* T_c, $c \in C$ *are scalarset types with special element* nil.
(ii) *For all unordered association ends* a, *the index type* τ_a *is a scalarset type without a special element.*[10]
(iii) *For all unordered behavioural features, e.g. operations,* f, *the index type* τ_f *is a scalarset type without a special element.* □

The proof of 24(i) is obvious for example by Lemma 19 since the formal semantics does not provide "forbidden" operations on the index types.[11] The proof of 24(ii) and (iii) cannot be obtained as directly. On the one hand, iterators over associations and behavioural features are not present in Θ and ρ because iteration is supposed to take multiple steps of the transition system in the semantics. But the loops are visible on a higher language level; thus the property of rule (S5) has to be checked on this higher level and then to be preserved by the preprocessing steps of [7].

[10] In order to be able to apply specific data-type reductions, it may be sensible to introduce *different* index types for different associations in different classes although they may be of the same multiplicity and hold the same type of references.

[11] Strictly speaking, the typing in the presentation of [7] and even in Section 3 is actually too weak to fulfil the syntactical criteria "as is": for brevity, both refer to O_C which is the union of all object reference types—although it is straightforward to obtain a representation which separates the object reference types s.t. we can obtain their scalarset property directly by syntactical criteria.

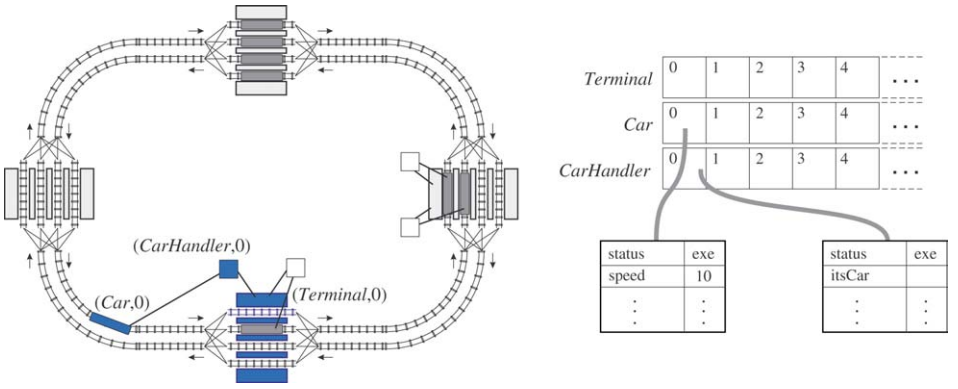


Fig. 5. **Query reduction:** The figure on the left shows, darkened, the objects (*Car*,0), (*Terminal*,0), and (*CarHandler*,0) that are referred to in the representative case. The figure on the right shows the same objects in a schematic diagram of the array of records that is introduced by [7] as one snapshot of the STS. The record values for (*Car*,0) and (*CarHandler*,0) are shown enlarged. Obviously, the state space is a priori not reduced: a state (or snapshot) still represents an unbounded number of objects. □

On the other hand we have to treat the fact that the variables written in different iterations of such a loop are not necessarily disjoint. In iteration over associations, for example, in order to iteratively send an event to all associated objects, it is the same queue into which events are inserted, yet it is symmetric as long as the events' parameters do not depend on the order of execution of iterations.

Given Lemma 24, Corollary 23 can be applied directly since according to Section 3, the semantics of an LSC w.r.t. a UML model is an LTL formula quantified over constants of types T_c. A representative set can be obtained as demonstrated in the proof of Lemma 22. By taking into account the activation condition, some of the representative cases may already be found to be trivially fulfilled leaving even fewer real model-checking tasks—for example if the specification contains two instances of the same type and the activation condition requires both to be different at activation time.

From a practical point of view, the verification tasks for the representative case do not depend on each other's results, so they may be carried out fully in parallel.

Going back to the example from the beginning of the section, we find that we refer to only one instance of classes *Car*, *Terminal*, and *CarHandler* in the LSC; thus the set

$$R =_{df} \{(\mathit{Car}, 0), (\mathit{Terminal}, 0), (\mathit{CarHandler}, 0)\} \subset C \times \mathbb{N}_0$$

is a (minimal) representative set (assuming $(c, 0) \neq \mathsf{nil}_c$ for classes $c \in C$). By the results of the previous section it is sufficient to verify only this single case of a concrete binding as illustrated by Fig. 5.

4.3. Model abstraction by data-type reduction

As depicted in Fig. 5, the query reduction does not affect the model, i.e. there is a priori no reduction of the model due to the application of query reduction. However, the representative proof obligations are *more specialised* than the whole property, that

is, each of these proof obligations tends to refer to a smaller part of the model. Thus the standard technique of cone-of-influence reduction (or "program slicing") can make some verification tasks practically feasible for all representative tasks for models where the whole property is not automatically verifiable for complexity reasons.

Unfortunately, the cone-of-influence reduction in general does not work well for UML models due to the indirect addressing of array places: if an object denoted in the property has a link to another object, then the value of this link is in general not restricted; thus all objects of the linked class are identified as being relevant for the property. Furthermore, an exact abstraction like the cone-of-influence reduction is only in special cases effective in reducing infinite state models to finite ones.

Thus, intuitively, we want to introduce an over-approximation in which only a finite set of, for example, ***Car*** objects is represented exactly and in which a new object identity $(\mathit{Car}, \perp)$ is introduced which denotes "one of the other, not exactly represented cars". From the viewpoint of an object of for example class ***Terminal*** with two links to ***Car*** objects it will not be distinguishable how many ***Car*** objects in addition to the exactly represented ones are alive in a particular snapshot.

An abstraction technique, which yields the desired result, is the *data-type reduction* introduced by McMillan [32]. It can be made explicit by a syntactical transformation of the system description modifying those places where any object refers to any other object: every occurrence of the comparison operator is replaced by a conditional expression which "guesses" a boolean value when both arguments have the value $\perp$; thus the transition relation comprises states representing *both* outcomes and thus has the properties sketched above. Every use of an expression of scalarset type as an array index is replaced by a conditional expression which yields an arbitrary value of the component type if the value of the index is $\perp$ and the original value otherwise.

Example 25. An expression which adds the value of the discretised ***speed*** attribute of two cars p and q

$$p \to \mathit{speed} + q \to \mathit{speed} = \mathit{Car}[(\mathit{Car}, p)].\mathit{speed} + \mathit{Car}[(\mathit{Car}, q)].\mathit{speed} \tag{3}$$

is changed to

$$\begin{aligned}&(p = \perp ?\, \mathit{guess}_1^{\mathit{Car}} : \mathit{Car}[(\mathit{Car}, p)]).\mathit{speed}\\ &\quad + (q = \perp ?\, \mathit{guess}_2^{\mathit{Car}} : \mathit{Car}[(\mathit{Car}, q)]).\mathit{speed}\end{aligned} \tag{4}$$

where τ is the type of attribute *speed* and 'guess_i^{τ}' are fresh free variables of type τ. It is obviously not sufficient to only change the domain of the unbounded array representing cars to a finite set where entry $(\mathit{Car}, \perp)$ is guessed anew in each step of a run, since then expression (3) would *always* yield the even number

$$2 \cdot p \to \mathit{speed} \tag{5}$$

if p and q both have value $\perp$ (since speed is discretised), although the abstraction is intended to also represent the case where p and q are different "other" cars which may well have different speeds. Expression (4) however can evaluate to every possible value allowed by the typing of attribute *speed*; thus it over-approximates the valuations observed in the original model. □

The remainder of this section is structured as follows. Section 4.3.1 defines data-type reduction on a scalarset type as a finite subset of the type's domain, presents a transformation on FOL expressions which implements the over-approximation, and establishes a relation between the states of the original and the abstract signature by means of a projection.

Section 4.3.2 applies the data-type reduction to the initial snapshot and transition predicate of an STS and shows that the abstract transition system simulates the concrete one and hence properties proven for the abstract transition system also hold for the concrete one. Section 4.3.3 briefly explicates the application of data-type reduction to parametrised systems.

Section 4.3.4 provides a straightforward heuristics for the application of data-type reduction to LSC verification for UML and "plays through" a scenario in the abstract system obtained from our running example which leads to a discussion of the problem of false-negatives, i.e. runs of the abstract system which contradict the property but are not valid runs of the original system. Section 4.3.5 discusses two directions for refining the abstraction in the case of false-negatives.

4.3.1. Data-type reduction

Definition 26 (*Data-type Reduction*). Let τ_s be a scalarset type. A subset $dtr_{\tau_s} \subseteq \mathcal{D}_{\tau_s} \setminus \mathsf{nil}(\tau_s)\}$ of its domain is called a *data-type reduction* (DTR) on τ_s.

Let $t = \{\tau \mid \tau \text{ type}\} \supseteq \{\tau_s\}$ be a set of types. A data-type reduction dtr_{τ_s} on τ_s induces a *data-type reduction 'dtr' on the domains of the (structured) types constructed from t* inductively as follows:

(i) If $\tau \in t$ is an unstructured type, then $dtr(\mathcal{D}_\tau) = dtr_{\tau_s} \cup \{\bot_{\tau_s}\} \cup \mathsf{nil}(\tau_s)\}$, if $\tau = \tau_s$, and $dtr(\mathcal{D}_\tau) = \mathcal{D}_\tau$ otherwise.
(ii) If $\tau = \tau_1 \times \cdots \times \tau_n$ is a record type, then $dtr(\mathcal{D}_\tau) = dtr(\mathcal{D}_{\tau_1}) \times \cdots \times dtr(\mathcal{D}_{\tau_n})$.
(iii) If $\tau = \tau_1 \to \tau_2$ is an array type, then $dtr(\mathcal{D}_\tau) = dtr(\mathcal{D}_{\tau_1}) \to dtr(\mathcal{D}_{\tau_2})$. □

If multiple data-type reductions for different scalarset types τ_s are applied, multiple distinct symbols $\bot_{\tau_s}$ representing "all other values (of type τ_s)" are introduced. We simply use $\bot$ when the context determines its type.

The following definition describes how we obtain a "data-type reduced expression" that implements the over-approximation as a prerequisite for data-type reduction of STSs. Therein we assume that the only constant (except for nil_{τ_s}) defined for a scalarset type τ_s is the test for equality extended onto the domain $dtr(\mathcal{D}_{\tau_s})$. If we allowed any other constant f defined on τ_s, then the over-approximation would be defined analogously using a conditional expression that guesses a value from the domain $\mathcal{M}_{type(f)}$ whenever at least *one* of the arguments evaluates to $\bot$.

Any other operation defined on τ_s would be over-approximated by introducing a conditional expression which would guess a value from τ_s's domain whenever at least *one* of the arguments evaluates to $\bot$.

Definition 27 (*Data-type Reduction for Expressions*). Let B be a signature, $\mathcal{M}$ a structure of B and τ_s a scalarset w.r.t. $\mathcal{M}$. Let dtr_{τ_s} be a data-type reduction on τ_s and $expr \in Expr_{FO}(B)$. The *data-type reduced expression* $dtr(expr)$ (or $expr_{dtr}$) $\in$

$B_{dtr} = (V_{dtr}, \Omega_{dtr})$ (as defined below) is obtained from '*expr*' by applying the following syntactical transformations:

(i) Two transformed expressions $expr_1$, $expr_2$ of the data-type reduced scalarset type τ_s which are compared for equality are changed s.t. the comparison yields a non-deterministic truth-value if both expressions have value $\perp$:

$$expr_1 = expr_2 \rightsquigarrow (expr_1 = \perp \wedge expr_2 = \perp \,?\, guess^{\mathbb{B}} : expr_1 = expr_2).$$

Set $expr_g \equiv_{df} expr_1 = expr_2$.

Every $guess^{\tau}$ stands for a different fresh variable $g \in V_{dtr} \setminus V$ of type τ. We set $expr_g \equiv_{df} expr_1 = expr_2$, i.e. $expr_g$ denotes the expression g was introduced for. All newly introduced g are collected in V_{dtr}.

(ii) Indexing an array $a : \tau_s \to \tau$ at index '*expr*' not on the left hand side of an "assignment" is changed s.t. it yields non-deterministically a value from a's component type τ if the index expression has value $\perp$:

$$a[expr] \rightsquigarrow (expr = \perp \,?\, guess^{\tau} : a[expr]).$$

$expr_g \equiv_{df} a[expr]$.

(iii) Indexing an array $a : \tau_s \to \tau$ at index $expr_1$ on the left hand side of an "assignment" (cf. [7]) is changed s.t. it is considered only if the index expression *does not* have value $\perp$:

$$a[expr_1]sel' := expr_2 \rightsquigarrow (expr_1 \neq \perp \implies a[expr_1]sel' := expr_2).$$

Here '*sel*' denotes a possibly empty *selection* if τ is a structured type. $expr_g \equiv_{df} a[expr_1]sel' := expr_2$.

Let $G = \{g_1, \ldots, g_n\}$ denote the set of all fresh variables introduced into $expr_{dtr}$ above. The signature B_{dtr} is $B_{dtr} = (V_{dtr}, \Omega_{dtr}) = (V \cup G, \Omega)$.

Let $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ be a structure of B. Then the canonical structure of B_{dtr} is $\mathcal{M}_{dtr} = (dtr(\mathcal{D}), dtr(\mathcal{I}))$ where $dtr(\mathcal{I})$ denotes the interpretation which coincides with $\mathcal{I}$ on $\Omega\backslash\{=\}$ and gives the natural interpretation on $dtr(\mathcal{D}_{\tau_s})$ to '=', i.e. does not consider $\perp$ to be special. □

The application of rule (iii) is not necessary; it is sound since the value of the $\perp$-th entry of a data-type reduced array a is never actually read, because every expression reading a is changed according to rule (ii) and thus never accesses $a[\perp]$. For the same reason there need not even be an actual storage place for the $\perp$-th entry of a data-type reduced array a.

Note that in the general case of Definition 27, a value from the component domain of an array is "guessed" which might be a large structure like in the UML semantics where it would guess the whole representation of an object. If parts of the structure are subsequently selected in the expression, a trivial optimisation consists of replacing the whole selection by "guessing" only a value of the selected part's type; in the example of the UML semantics only the value of a navigated attribute would be guessed.

An extension of Definition 27 considering loops over unbounded associations would have to account for the fact that in the form of the length of the queue, there is even a place which indirectly *counts* the number of associated objects when sending an event to each

associated object. Thus in order to provide a proper over-approximation, the iteration over associated objects has to guess whether the iteration is supposed to terminate and to keep track of whether all concrete indices have been visited.

4.3.2. Simulation

Given an STS over a signature with a scalarset type, the data-type reduced STS is obtained as follows:

Definition 28 (*Data-type Reduced STS*). Let $S = (B, \Theta, \rho)$ be an STS, $\mathcal{M}$ a structure of B, $\tau_s \in T_B$ a scalarset, and dtr_{τ_s} a data-type reduction on τ_s. Then the *data-type reduced* STS induced by dtr_{τ_s} is

$$dtr(S) =_{df} S_{dtr} =_{df} (B_{dtr}, \Theta_{dtr}, \rho_{dtr}). \quad \square$$

The following lemma claims that the data-type reduced system is a simulation of the original system in the sense of Definition 29 below, i.e. for every run in the original system there is a related run in the abstract system. Intuitively, the abstracted initiation and transition predicates accept more states as initial and more pairs of states to be S-successors than the original predicates.

Definition 29 (*Simulation*). Let $S_1 = (B_1, \Theta_1, \rho_1)$ and $S_2 = (B_2, \Theta_2, \rho_2)$ be STSs, $\mathcal{M}_1$ and $\mathcal{M}_2$ structures of their signatures, and Σ_1 and Σ_2 the sets of their snapshots.

We say S_1 *simulates* S_2, $S_1 \rightsquigarrow S_2$, iff there exists a relation $\varrho \subseteq \Sigma_1 \times \Sigma_2$ s.t.

(i) $\forall\, s_2 \in \Sigma_2 : \mathcal{M}_2[\![\Theta_2]\!](s_2) \implies \exists\, s_1 \in \Sigma_1 : \mathcal{M}_1[\![\Theta_1]\!](s_1) \wedge (s_1, s_2) \in \varrho$
(ii) $\forall\, (s_1, s_2) \in \varrho\ \forall\, s_2' \in \Sigma_2 :$

$$\mathcal{M}_2[\![\rho_2]\!](s_2, s_2') \implies \exists\, s_1' \in \Sigma_1 : \mathcal{M}_1[\![\rho_1]\!](s_1, s_1') \wedge (s_1', s_2') \in \varrho.$$

The relation ϱ is called a *simulation relation*. $\square$

The following notion of a projection from concrete onto data-type reduced states plays a central role in the subsequent Lemma 31 which claims that the concrete system is simulated by the abstract one.

Definition 30 (*Projection*). Let $B = (V, \Omega)$ be a signature, $\mathcal{M}$ a structure of B, $\tau_s \in T_B$ a scalarset w.r.t. $\mathcal{M}$, and dtr_{τ_s} a data-type reduction on τ_s. Let $B_{dtr} = (V_{dtr}, \Omega_{dtr})$ be the signature as obtained from Definition 27 and $s \in \Sigma_B$ a valuation of the variables in V.

The projection of s onto dtr, $\downarrow_{dtr}(s) : \Sigma_B \to \Sigma_{B_{dtr}}$, is defined inductively as follows:

(i) If $v \in V$ is a variable of basic type τ, then

$$\downarrow_{dtr}(s)(v) = \begin{cases} \bot & \text{, if } \tau = \tau_s \text{ and } s(v) \notin dtr \\ s(v) & \text{, otherwise.} \end{cases}$$

(ii) If $v \in V$ is a variable of record type $\tau_1 \times \cdots \times \tau_n$ and $s(v) = (x_1, \ldots, x_n)$, then $\downarrow_{dtr}(s)(v) = (\downarrow_{dtr}(s)(x_1), \ldots, \downarrow_{dtr}(s)(x_n))$.
(iii) If $v \in V$ is a variable of array type $\tau_1 \to \tau_2$, then $\downarrow_{dtr}(s(v)) = \{i \mapsto \downarrow_{dtr}(s(v)(i))\}$.
(iv) If $v = g \in V_{dtr} \setminus V$, then $\downarrow_{dtr}(s(v)) = \downarrow_{dtr}(\mathcal{M}[\![expr_g]\!](s))$. $\square$

Lemma 31 (*DTR Simulation*). *Let $S = (B, \Theta, \rho)$ be an STS, $\mathcal{M}$ a structure of B, $\tau_s \in T_B$ a scalarset w.r.t. $\mathcal{M}$, dtr a data-type reduction on τ_s, and S_{dtr} the data-type reduced STS. Let $\mathcal{M}_{dtr}$ be the canonical structure of B_{dtr}. Then $S_{dtr} \rightsquigarrow S$.* □

Proof. Set $\varrho =_{df} \{(s, \downarrow_{dtr}(s)) \mid s \in \Sigma\}$. ϱ is a simulation relation:

(i) Let $s_2 \in \Sigma$ s.t. $\mathcal{M}[\![\Theta]\!](s_2) = true$.
Choose $s_1 = \downarrow_{dtr}(s_2)$. Then $(s_1, s_2) \in \varrho$ and $\mathcal{M}[\![\Theta_{dtr}]\!](s_1) = true$ by induction over the structure of Θ_{dtr}:

- Let s be a valuation of variables in V. The data-type reduction affects only the following cases:
  - $\Theta \equiv expr_1 = expr_2$, both $expr_1, expr_2$ of type τ_s:
    Then Θ has been changed to
    $$\Theta_{dtr} \equiv (expr_1 = expr_2 = \bot\, ?\, guess^{\mathbb{B}} : expr_1 = expr_2),$$
    where $guess^{\mathbb{B}}$ denotes a variable $g \in V_{dtr}$; thus
    $$\begin{aligned}\mathcal{M}[\![\Theta_{dtr}]\!](\downarrow_{dtr}(s)) &= \mathcal{M}[\![(expr_1 = expr_2 = \bot\, ?\, g : expr_1 = expr_2)]\!](\downarrow_{dtr}(s)) \\ &= \mathcal{M}[\![expr_1 = expr_2]\!](s).\end{aligned}$$
  - $\Theta \equiv a[expr_1]$, with $expr_1$ an expression of type τ_s and a with components of boolean type:
    Then Θ has been changed to
    $$\Theta_{dtr} \equiv (expr_1 = \bot\, ?\, guess^{\mathbb{B}} : a[expr_1]),$$
    where $guess^{\mathbb{B}}$ denotes a variable $g \in V_{dtr}$; thus
    $$\begin{aligned}\mathcal{M}[\![\Theta_{dtr}]\!](\downarrow_{dtr}(s)) &= \mathcal{M}[\![(expr_1 = \bot\, ?\, g : a[expr_1])]\!](\downarrow_{dtr}(s)) \\ &= \mathcal{I}(?:)(\mathcal{M}[\![expr_1]\!](\downarrow_{dtr}(s)) = \bot, \\ &\qquad \downarrow_{dtr}(s)(g), \mathcal{M}[\![a[expr_1]]\!](\downarrow_{dtr}(s))) \\ &= \mathcal{M}[\![a[expr_1]]\!](s)\end{aligned}$$
    since $\mathcal{M}[\![expr_1]\!](\downarrow_{dtr}(s)) = \bot$ implies $\downarrow_{dtr}(s)(g) = \mathcal{M}[\![a[expr_1]]\!](s)$ by construction of $\downarrow_{dtr}(s)$.
    And analogously for structured array values from which a component of boolean type is selected.

(ii) Let $(s_1, s_2) \in \varrho$ and $s_2' \in \Sigma$ s.t. $\mathcal{M}[\![\rho]\!](s_2, s_2') = true$.
Choose $s_1' = \downarrow_{dtr}(s_2')$. Then $(s_1, s_2) \in \varrho$ and $\mathcal{M}[\![\rho_{dtr}]\!](s_1, s_1') = true$ by induction over the structure of ρ_{dtr}, analogously to (i). □

Now Lemma 31 provides us with a run of the abstract transition system for each run of the original transition system in the following.

Lemma 32 (*Data-type Reduction*). *Let $S = ((V_0, \Omega_0), \Theta_0, \rho_0)$ be an STS, $\mathcal{M}_0$ a structure of (V_0, Ω_0), and $\tau_s \in T_{B_0}$ a scalarset w.r.t. $\mathcal{M}_0$. Let X be a finite set of 0-ary constants and ϕ an LTL formula over $B = (V, \Omega)$ with $X \subseteq \Omega$ s.t. $B \subseteq_X (V_0, \Omega_0)$. Let dtr_{τ_s} be a data-type reduction on the scalarset type τ_s and S_{dtr} the data-type reduced STS*

induced by dtr_{τ_s}. *Let* $\mathcal{M}_{dtr}$ *be the canonical structure of* S_{dtr}. *Then*

$$S_{dtr} \models_{\mathcal{M},\forall(,0)} \phi \implies S \models_{\mathcal{M},\forall(,0)} \phi \text{ and}$$
$$S \models_{\mathcal{M},\exists(,0)} \phi \implies S_{dtr} \models_{\mathcal{M},\exists(,0)} \phi. \quad \square$$

Proof. (By contraposition.)

(i) Let $S \not\models_{\mathcal{M},\forall} \phi$, i.e. $\exists\, r \in runs(S)\ \exists\, i \in \mathbb{N}_0 : r/i \not\models_{\mathcal{M}} \phi$. By (the proof of) Lemma 31, there exists a run $r_{dtr} \in runs(S_{dtr})$ s.t. $\forall\, i \in \mathbb{N}_0 : r^i_{dtr} = dtr(r^i)$.
Then $S_{dtr} \not\models_{\mathcal{M},\forall} \phi$ follows by induction over the structure of ϕ:
- $\phi \equiv expr$:
  Then $\mathcal{M}[\![expr]\!](r^i) = false$. Analogously to the proof of Lemma 31 by induction over the structure of $expr$, $\mathcal{M}[\![expr]\!](r^i_{dtr}) = false$; thus $r_{dtr}/i \not\models_{\mathcal{M}} \phi$.
- $\phi \equiv f \vee g$: Then $r/i \not\models_{\mathcal{M}} f$ and $r/i \not\models_{\mathcal{M}} g$.
  By the induction hypothesis, $r_{dtr}/i \not\models_{\mathcal{M}} f$ and $r_{dtr}/i \not\models_{\mathcal{M}} g$.
- $\phi \equiv \neg f$: Then $r/i \models_{\mathcal{M}} f$. By the induction hypothesis $r_{dtr}/i \models_{\mathcal{M}} f$.
- $\phi \equiv \mathbf{X}\, f$: Then $r/i+1 \not\models_{\mathcal{M}} f$. By the induction hypothesis $r_{dtr}/i+1 \not\models_{\mathcal{M}} f$.
- $\phi \equiv \mathbf{G}\, f$: Then $\exists\, j \in \mathbb{N}_0 : r/i+j \not\models_{\mathcal{M}} f$.
  By the induction hypothesis $r_{dtr}/i+j \not\models_{\mathcal{M}} f$; thus $r_{dtr}/i \not\models_{\mathcal{M}} \phi$.
- $\phi \equiv f\ \mathbf{U}\ g$: Distinguish two cases:
  (a) *Not finally* g: $\forall\, j \in \mathbb{N}_0 : r/i+j \not\models_{\mathcal{M}} g$. Then by the induction hypothesis $\forall\, j \in \mathbb{N}_0 : r_{dtr}/i+j \not\models_{\mathcal{M}} g$; thus $r_{dtr}/i \not\models_{\mathcal{M}} \phi$.
  (b) *Not* f *before* g: For each $j \in \mathbb{N}_0$ s.t. $r/i+j \models_{\mathcal{M}} g$, there exists $0 \le k < j$ s.t. $r/i+k \not\models_{\mathcal{M}} f$. Let $j' \in \mathbb{N}_0$ be s.t. $r_{dtr}/i+j' \models_{\mathcal{M}} g$. Then by the induction hypothesis there exists $k' < j'$ s.t. $r_{dtr}/i+k' \not\models_{\mathcal{M}} f$; thus $r_{dtr}/i \not\models_{\mathcal{M}} \phi$.

And similarly for $\models_{\mathcal{M},\forall,0}$.

(ii) The cases $\models_{\mathcal{M},\exists}$ and $\models_{\mathcal{M},\exists,0}$ are obtained directly by construction of S_{dtr}. $\square$

The restriction put on ϕ in the premise of Lemma 32 is not as strong as it may seem since a boolean expression with a subterm $expr_0$ of type τ_s can easily be integrated into the model as a boolean auxiliary variable, i.e. a boolean variable which is assigned the value of $expr_0$ in each step and not used otherwise. Then ϕ can be transformed into a ϕ' which refers to the auxiliary variable instead of $expr_0$ and thus satisfies the premise of Lemma 32. Within the model, $expr_0$ undergoes the changes of Definition 27.

Note that for formal verification, only the former implication of Lemma 32 is of practical relevance: if we are able to prove a property for the abstract system, then it holds in the original system. But if we are seeking for an example run in order to verify an existential formula, there is no guarantee that a run found in the abstract system is also a run of the concrete system.

4.3.3. Parametrised designs

A direct corollary of the previous Section 4.3.2 is the following [32]:

Corollary 33. *Let* $S = ((V_0, \Omega_0), \Theta_0, \rho_0)$ *be an STS,* $\mathcal{M}_0$ *a structure of* (V_0, Ω_0), *and* $\tau_s \in T_{B_0}$ *a scalarset w.r.t.* $\mathcal{M}_0$. *Let* X *be a finite set of 0-ary constants and* ϕ *an LTL formula over* $B = (V, \Omega)$ *with* $X \subseteq \Omega$ *s.t.* $B \subseteq_X (V_0, \Omega_0)$. *Let* dtr_{τ_s} *be a data-type*

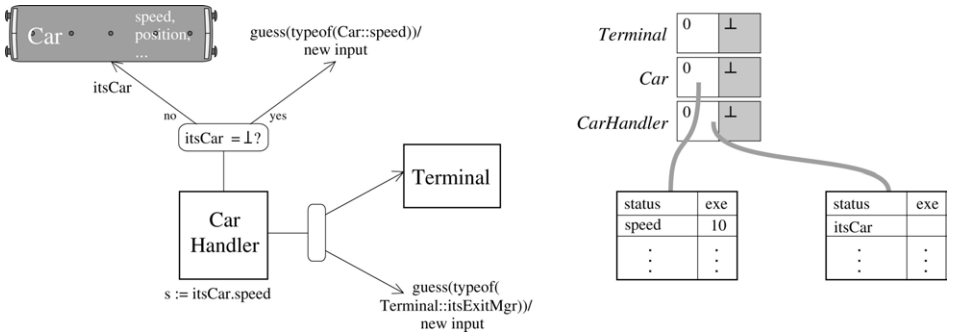


Fig. 6. **Data-type reduction:** Intuitively, data-type reduction modifies the view of the concrete objects onto the outside world at each place in the behaviour description where the outside world is referred to. On the left we show a schematic diagram of a car handler object that has in the concrete model a link to a car and a terminal. The abstract interpretation can be seen as cutting this link and, when reading it, letting the read pass through for concrete object identities and redirecting the read to a fresh input that yields an arbitrary value. Hence a concrete car handler cannot determine the number of car objects alive. On the right we show a schematic diagram of the resulting state space analogous to the one shown in Fig. 5, but providing those indices with index $\perp$ only shaded since they need not actually be represented. The resulting transition system is finite state if all data-types and queues are finite. □

reduction on the scalarset type τ_s, and S_{dtr} the data-type reduced STS induced by dtr_{τ_s}. Let $\mathcal{M}_{dtr}$ be the canonical structure of S_{dtr} and let dtr' be a data-type reduction on τ_s s.t. $dtr' \supseteq dtr$. Then

$$S_{dtr} \models_{\forall(,0)} \phi \implies S_{dtr'} \models_{\forall(,0)} \phi. \quad \square$$

That is, if a property ϕ holds for *some* data-type reduced system, then it holds for *each* larger system. This reformulation of Lemma 32 is in particular relevant for parametrised systems like the *ARCS* since it provides a technique for proving properties for all instantiations of a parametrised model.

4.3.4. Data-type reduction for UML

In the domain of LSC verification for UML, data-type reduction applies directly to the tasks obtained from query reduction. Following the methodology proposed by McMillan [32] the initial data-type reduction is obtained from the task by heuristics:

Let M be a UML model and L a (for instance universal) LSC with instance lines annotated by N different class types $T_{c_1}, \ldots, T_{c_N}$. According to Definition 10, a system S satisfies L if it satisfies all possible bindings of object identifiers to constants in the resulting LTL formula where each constant directly corresponds to an instance line. By results of Section 4.2, it is sufficient to consider a finite representative set of bindings. Consider one of these bindings and let $i_{k_1}, \ldots, i_{k_n} \in T_{c_k}$, $1 \leq k \leq N$, be the chosen concrete objects of type T_{c_k}, i.e. separated by type. Then we first try to verify the task on the data-type reduced system

$$dtr(S) = dtr_m(\ldots(dtr_{N+1}(dtr_N(\ldots(dtr_1(S))\ldots)))\ldots)$$

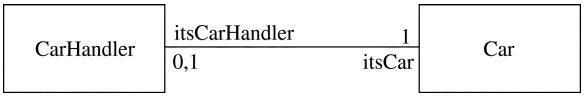


Fig. 7. **Class diagram:** The relation between *Car* and *CarHandler*. □

where we choose as data-type reductions $dtr_k = \{o_{k_1}, \ldots, o_{k_n}\}$ for $1 \leq k \leq N$, i.e. each type used in the specification, and $dtr_k = \emptyset$ for $N < k \leq m$ if the UML model comprises m class types overall.

Example 34. For the running example, we would apply the data-type reductions,

$$dtr_{Car} =_{df} \{(Car, 0)\}, dtr_{Terminal} =_{df} \{(Terminal, 0)\},$$
$$dtr_{CarHandler} =_{df} \{(CarHandler, 0)\},$$

and yield a system as illustrated by Fig. 6.

To illustrate the effect of the abstraction on the observable dynamic behaviour, we briefly "play through" event sending and operation calls in the following.

Consider class *Car* in the *ARCS* which has an association to a *CarHandler* (cf. Fig. 7). Executing the create action in the *Terminal* means [7] selection of one of the as yet unused, non-alive objects from the set of all objects. In the abstract system, the alive flag of the concrete objects is considered as well as the alive flag obtained by the reference $\perp_{CarHandler}$. Thus by executing the create action, the *Terminal* may get a reference to a concrete object or to "any other *CarHandler*". This actually provides for unbounded creation of *CarHandler*s: after all concrete objects have become alive, all subsequent (possibly infinitely many) create operations yield the object identity $\perp_{CarHandler}$. In the following, we assume that the $\perp_{CarHandler}$ is indeed returned to the *Terminal* in response to the create action and passed to the *Car*.

When the *Car* sends an event to this *CarHandler*, the event is entered into the event queue of the active object responsible for the *CarHandler*. In this case the identity of the active object is actually guessed since the reference to the responsible active object is an (inherited) attribute of *CarHandler* and as such just guessed for $\perp_{CarHandler}$. Thereby, the event may end up in *any* event queue. In the event queue of a concrete active object, the event will move to the top of the queue and become ready to be dispatched.

At dispatch time, the changed transition predicate causes the current *state* of the destination, *CarHandler* $\perp_{CarHandler}$, to be *guessed*. Thus the event may be discarded or accepted. If the choice is for acceptance, the corresponding active object notes down $\perp_{CarHandler}$ as the currently processing object and as long as the predicate $stable(\perp_{CarHandler})$, which indicates the end of a run-to-completion step [7], is not evaluated to true, all possible actions of the state-machine of class *CarHandler* may be executed.

Whenever during the execution of actions the associations and attributes of $\perp_{CarHandler}$ are evaluated, the value is in fact guessed; hence for example the transition which sends an event back to the *Car* managed by the *CarHandler* might choose any object of class *Car* in the system as destination: an unwanted interference (cf. Section 4.3.5).

Analogously, when a *Car* calls a triggered operation of $\perp_{CarHandler}$, the object reference $\perp_{CarHandler}$ is entered into the *Car*'s pending request table as the receiver. The transformed transition predicate of the abstract system also processes this entry for $\perp_{CarHandler}$, but again it is possible to execute arbitrary transitions or becoming stable since, for example, the current state of $\perp_{CarHandler}$ is simply guessed. When the choice happens to become stable, the pending request table entry is changed to '*completed*' and the caller may continue. If there were a reply action within the sequence of actions executed by the callee, then the caller might find any possible value as the reply if the reply depended on attributes of the callee. Note that the callee $\perp_{CarHandler}$ in particular need not become stable; thus we can observe any number of steps between the call of the triggered operation and its completion.

In general, the proposed abstraction technique is obviously not well suited for liveness properties which is indeed a slight mismatch to our specification language of *live* sequence charts, yet it applies well to all safety properties expressed as LSC. □

4.3.5. Abstraction refinement, interference, and non-interference lemmata

The heuristics for the initial data-type reduction as presented in the previous section yields a very coarse abstraction—in particular since for those classes which do not occur in the LSC, there is not a single concretely represented object in the abstracted system. For example, in the *ARCS* an LSC which requires that two cars do not collide may comprise only two instance lines at first, one for each car. But since the position of a car depends on its speed which is measured by its cruiser, there will be a false-negative if all cruiser objects are abstracted according to the heuristic data-type reduction.

In order to refine the abstraction we would add cruiser instance lines to the LSC and relate cars and cruisers to each other by the activation condition. At first, the LSC need not necessarily comprise any communication between cars and cruisers. We only make this dependency of the requirement on cruisers explicit.[12] Then the heuristics is applied to the new LSC that now comprises some cruisers and it will yield a system with as many concrete cruisers as are needed for the cars. If the property again does not hold, the procedure is iterated.

The discussion of event sending at the end of the previous section already named the other typical reason for false-negatives: "any other *CarHandler*" may send an event to a *Car* although it is not *the CarHandler* known by the *Car* (cf. Fig. 8). In the original system, only a single *CarHandler* actually knows a *Car* and sends events only when appropriate. In his SMV tutorial [30], K.L. McMillan demonstrates how to address this problem via so called *non-interference lemmata*. A non-interference lemma is a property that can be generalised to the following form:

"If some entity sends something to me, then it is allowed to do so".

The lemma is verified in a separate proof and taken as an *assumption* in the proof of the main property to rule out unwanted interferences.

[12] This is the LSC equivalent to "*case splitting*" in the methodology of McMillan [32].

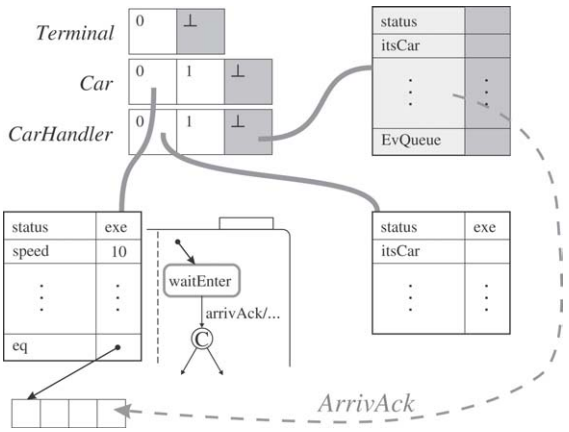


Fig. 8. **False-negative:** When checking the property "two *Cars* will not crash when entering the *Terminal* since their *CarHandler*s set the switches safely", the heuristics of Section 4.3 yields the depicted system. As discussed in Section 4.3.4, "any other *CarHandler*" sending an event *arrivAck* to the concrete *Car* awaiting this event could cause the *Car* to enter the terminal although all platforms are already occupied. □

In general, it might be necessary to explicitly introduce *auxiliary variables* which keep track of "the ones who are allowed to send". But in UML models, binary associations a with association ends e_1, e_2, each of multiplicity 0..1 or 1, are often intended to be "bi-directional",[13] i.e. the navigation forth and back along the association yields the identity: $self.e_1.e_2 = self$. If furthermore communication in the model is closely related to associations in the sense that an event's destination is always given in terms of an association, and if the modeller provides annotations of all such associations with a set of signals, which are intended to be sent "along" this association, we can heuristically derive the non-interference lemma: sending an event to e_2 implies $e_2.e_1 = self$.

In the above example, there exists only the single association between class *Car* and class *CarHandler* depicted in Fig. 7. Thus we would derive

$$\forall\, h \in O_{CarHandler}\, \forall\, c \in O_{Car}:$$
$$justsend = (true, h, c, \ldots) \implies h \to itsCarHandler = c$$

which has to be proven separately. Note that again all symmetry reduction and abstraction techniques apply to this property, and may indeed be needed due to the infinite state space.

5. Conclusion

We have provided an extension of the LSC semantics which explains dynamic binding of system entities to instance lines by interpreting them as quantifications over object identifiers and given meaning to message sending and reception by a mapping for UML models in the semantics of [7]. In particular it allows one to refer to object creation and

[13] Although this interpretation is (reasonably) not enforced by the UML 2.0 proposals.

destruction during the activation of an LSC and thus to denote objects which are not existing at activation time.

In order to formally verify an LSC against the in general infinite state UML model by automatic finite state methods, we proposed a two-step approach which is basically a transfer of the methodology of [32] to the UML domain. We first observe that the types of object identities and association indices within the UML model in the semantics of [7] are symmetric types that induce a symmetric transition relation and hence all infinitely many possible bindings of objects to instance lines are implied by a finite set of representative cases. Secondly, for each of the representative cases, the abstraction technique of data-type reduction is applied to the object reference and association index types. This technique yields a finite over-approximation of the original system behaviour where false-negatives are treated by using less coarse abstractions and by automatically proving non-interference lemmata derived from information in the UML model.

Acknowledgements

This research was partially supported by the German Research Council (DFG) within the priority programme Integration of Specification Techniques with Engineering Applications under grant DA 206/7-3 and by the Information Society DG of the European Commission within the project IST-2001-33522 OMEGA (Correct Development of Real-Time Embedded Systems).

An abridged version appeared in *Formal Methods for Components and Objects, 1st Int. Symp., FMCO 2002, LNCS 2852, Springer, 2003, pp. 99–135*.

We would like to thank T. Toben for proofreading and listening, and the anonymous reviewers of both this and the abridged version for valuable comments and suggestions.

References

[1] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, The MIT Press, 1999.

[2] J. Corbett, M. Dwyer, J. Hatcliff, Robby, a language framework for expressing checkable properties of dynamic software, in: K. Havelund, J. Penix, W. Visser (Eds.), SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30–September 1, 2000, Proceedings, LNCS, vol. 1885, Springer-Verlag, 2000, pp. 205–223.

[3] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, D. Varró, Viatra—visual automated transformations for formal verification of UML models, in: W. Emmerich, D. Wile (Eds.), ASE, IEEE Computer Society, 2002.

[4] W. Damm, D. Harel, LSCs: breathing life into message sequence charts, Formal Methods in System Design 19 (1) (2001) 121–141.

[5] W. Damm, B. Jonsson, Eliminating queues from RT UML model representations, in: Damm and Olderog [6], pp. 375–394.

[6] W. Damm, E.-R. Olderog (Eds.), Formal techniques in real-time and fault-tolerant systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9–12, 2002, Proceedings, LNCS, vol. 2469, Springer-Verlag, 2002.

[7] W. Damm, B. Josko, A. Pnueli, A. Votintseva, A discrete-time UML semantics for concurrency and communication in safety-critical applications, Science of Computer Programming (in this issue) doi:10.1016/j.scico.2004.05.012.

[8] A. David, M.O. Möller, W. Yi, Formal verification of UML statecharts with real-time extensions, in: R.-D. Kutsche, H. Weber (Eds.), Fundamental Approaches to Software Engineering, FASE'2002, LNCS, vol. 2306, Springer-Verlag, 2002, pp. 218–232.

[9] D. Distefano, On model checking the dynamics of object-based software, Ph.D. Thesis, University of Twente, 2003.
[10] E.A. Emerson, A.P. Sistla, Symmetry and model checking, Formal Methods in System Design 9 (1–2) (1996) 105–131.
[11] M. Feather, M. Goedicke (Eds.), 16th IEEE International Conference on Automated Software Engineering, ASE 2001, 26–29 November 2001, Coronado Island, San Diego, CA, USA, IEEE Computer Society, 2001.
[12] A.L. Guennec, Genie logiciel et methodes formelles avec UML—specification, validation et generation de tests, Ph.D. Thesis, Université de Rennes 1, 2001.
[13] D. Harel, E. Gery, Executable object modeling with statecharts, IEEE Computer 30 (7) (1997) 31–42.
[14] D. Harel, R. Marelly, Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine, Springer-Verlag, 2003.
[15] R. Iosif, Symmetry reduction criteria for software model checking, in: D. Bosnacki, S. Leue (Eds.), 9th International SPIN Workshop, Grenoble, France, April 11–13, 2002, Proceedings, LNCS, vol. 2318, Springer-Verlag, 2002, pp. 22–41.
[16] R. Iosif, Exploiting heap symmetries in explicit-state model checking of software, in: Feather and Goedicke [11].
[17] C.N. Ip, D.L. Dill, Better verification through symmetry, Formal Methods in System Design 9 (1–2) (1996) 41–75.
[18] C.N. Ip, D.L. Dill, Verifying systems with replicated components in Murφ, Formal Methods in System Design 14 (3) (1999) 273–310.
[19] ITU-T, ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1993.
[20] ITU-T, ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1996.
[21] ITU-T, ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1999.
[22] J. Klose, Live sequence charts: a graphical formalism for the specification of communication behavior, Ph.D. Thesis, Carl von Ossietzky Universität Oldenburg, 2003.
[23] J. Klose, B. Westphal, Relating LSC specifications to UML models, in: H. Ehrig, M. Grosse-Rhode (Eds.), Proceedings INT 2002—International Workshop on Integration of Specification Techniques for Applications in Engineering, 2002.
[24] J. Klose, H. Wittke, An automata based interpretation of live sequence charts, in: T. Margaria, W. Yi (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001, Proceedings, LNCS, vol. 2031, Springer-Verlag, 2001, pp. 512–527.
[25] A. Knapp, S. Merz, C. Rauh, Model checking timed UML state machines and collaborations, in: Damm and Olderog [6], pp. 395–416.
[26] D. Latella, I. Majzik, M. Massink, Automatic verification of a behavioral subset of UML statechart diagrams using the SPIN model-checker, Formal Aspects of Computing 11 (6) (1999) 637–664.
[27] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, New York, 1991.
[28] R. Marelly, D. Harel, H. Kugler, Multiple instances and symbolic variables in executable sequence charts, Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4–8, 2002, ACM, SIGPLAN Notices 37 (11) (2002) 83–100.
[29] K.L. McMillan, Verification of an implementation of Tomasulo's algorithm by compositional model checking, in: A.J. Hu, M.Y. Vardi (Eds.), Computer Aided Verification, 10th International Conference, CAV'98, Vancouver, LNCS, vol. 1427, Springer-Verlag, 1998, pp. 110–121.
[30] K.L. McMillan, Getting started with SMV, Tech. Rep., Cadence Berkeley Labs, http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial.ps, March 1999.
[31] K.L. McMillan, Verification of infinite state systems by compositional model checking, in: L. Pierre, T. Kropf (Eds.), Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME'99, LNCS, vol. 1703, Springer-Verlag, 1999, pp. 219–233.
[32] K.L. McMillan, A methodology for hardware verification using compositional model checking, Science of Computer Programming 37 (2000) 279–309.
[33] I. Ober, M. Bozga, Adapting and optimizing existing timed model checking tools to UML tools, Tech. Rep. IST/33522/WP2.1/D2.1.2, Verimag, December 2002.

[34] OMG, OMG Unified Modeling Language Specification, version 1.4 (September 2001).
[35] I. Paltor, J. Lilius, Formalising UML state machines for model checking, in: R.B. France, B. Rumpe (Eds.), UML'99: The Unified Modeling Language—Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28–30, 1999, Proceedings, LNCS, vol. 1723, Springer-Verlag, 1999, pp. 430–445.
[36] T. Schäfer, A. Knapp, S. Merz, Model checking UML state machines and collaborations, Electronic Notes in Theoretical Computer Science 55 (3) (2001).
[37] R.C. Schlör, Symbolic timing diagrams: a visual formalism for model verification, Ph.D. Thesis, Carl von Ossietzky Universität Oldenburg, January 2000.
[38] W. Shen, K. Compton, J.K. Huggins, A toolset for supporting UML static and dynamic model checking, in: Feather and Goedicke [11], pp. 315–318.
[39] F. Xie, J.C. Browne, Integrated state space reduction for model checking executable object-oriented software system designs, in: R.-D. Kutsche, H. Weber (Eds.), FASE, Lecture Notes in Computer Science, vol. 2306, Springer, 2002.
[40] F. Xie, V. Levin, J.C. Browne, Model checking for an executable subset of UML, in: Feather and Goedicke [11].
[41] E. Yahav, T. Reps, S. Sagiv, R. Wilhelm, Verifying temporal heap properties specified via evolution logic, in: P. Degano (Ed.), Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7–11, 2003, Proceedings, LNCS, vol. 2618, Springer-Verlag, 2003, pp. 204–222.